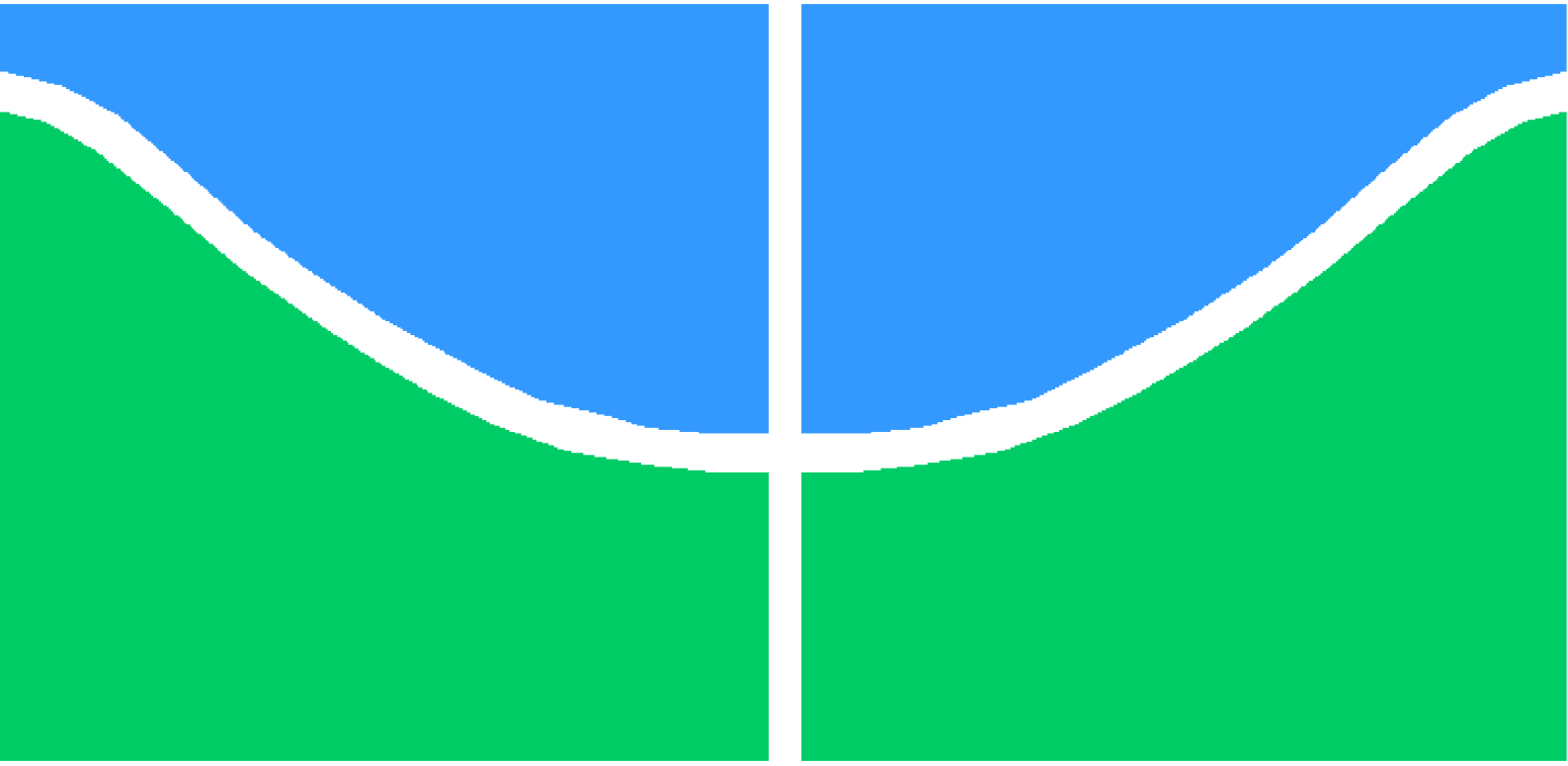

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

FGAme

Autor: Gustavo Cavalcante Oliveira
Orientador: Dr. Fábio Macêdo Mendes

Brasília, DF
2018

Gustavo Cavalcante Oliveira

FGAme

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Dr. Fábio Macêdo Mendes

Brasília, DF

2018

Gustavo Cavalcante Oliveira
FGAme/ Gustavo Cavalcante Oliveira. – Brasília, DF, 2018-
77 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Fábio Macêdo Mendes

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2018.

1. engine. 2. física. I. Dr. Fábio Macêdo Mendes. II. Universidade de Brasília.
III. Faculdade UnB Gama. IV. FGAme

CDU 02:141:005.6

Gustavo Cavalcante Oliveira

FGAme

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 14 de fevereiro de 2018 – Data da aprovação do trabalho:

Dr. Fábio Macêdo Mendes
Orientador

Dr. Edson Alves da Costa Júnior
Convidado 1

Dra. Carla Silva Rocha Aguiar
Convidado 2

Brasília, DF
2018

Agradecimentos

Primeiramente gostaria de agradecer à todos os amigos que fiz durante a graduação, vocês fizeram da UnB um local mais agradável e também aos meus amigos de longa data pelo apoio fora da universidade.

Gostaria de agradecer à todos os professores que fizeram parte desta jornada, em especial ao professor George Marsicano pelos ensinamentos passados durante a minha passagem pelo projeto Fábrica de Software, ao professor Edson Alves por inspirar os alunos nas aulas de TEP e ao professor Fábio Mendes pela orientação neste trabalho.

Resumo

A FGAme é uma *game engine* criada na Universidade de Brasília para ser utilizada na disciplina de Física para Jogos que sendo que a mesma é focada na criação de jogos educacionais, visando ser bastante simples de se utilizar com um motor de física já embutido. Este trabalho tem como objetivo apresentar a FGAme a comunidade acadêmica, mostrando conceitos da Engenharia de Software que foram aplicados e também na implementação de novas funcionalidades visando evoluir o *software*.

Palavras-chaves: Motor de Jogo. Física. Jogos Educacionais

Abstract

FGAme is a game engine created in the University of Brasília to be used in the physics for eletronic game class focused on the creation of educational games seeking to be very simple to use. This work has as objective to present the FGAme to the academic community showing the concepts of Software Engineering that has been applied and also the implementation of new features aiming to evolve this software.

Key-words: game engine. educational games. Android.

Lista de ilustrações

Lista de tabelas

Sumário

Introdução

Contextualização

O termo *game engine* pode ser usado para representar um *software* que é extensível e serve como base para diferentes jogos eletrônicos sem ser muito modificado (JASON, 2009), ou seja, uma *engine* funciona como um *software* que auxilia na criação de jogos, fazendo com que o programador de um jogo foque principalmente na lógica do jogo específico e reutilize códigos que são comuns a diferentes jogos.

A FGAme é uma *engine* de jogos 2d focada na simulação de física e facilidade de uso, criada pelo Professor Fábio Mêndes da Faculdade do Gama da Universidade de Brasília. O foco da *engine* é ser de fácil utilização para que até uma pessoa que esteja aprendendo a programar consiga utilizá-la, pensando nisso, ela foi escrita utilizando a linguagem de programação Python, visto que é uma linguagem que possui uma curva de aprendizado muito menor que a linguagem C++, que é bastante utilizada no contexto de jogos. Para qualquer pessoa que está aprendendo a programar é importante focar nos conceitos de programação e não nos conceitos específicos de cada linguagem (BOGDANCHIKOV et al., 2013).

Esse trabalho tem como foco apresentar a FGAme à comunidade, implementação de funcionalidades e correção de defeitos. Portanto, não se trata de um trabalho de pesquisa, mas sim de um relatório onde visa deixar claro à aplicação dos conceitos da Engenharia de Software em um *software open source.*

Objetivos

Objetivo geral

O objetivo geral deste trabalho é evoluir o código da FGAme e implementar práticas da Engenharia de Software na FGAme.

Objetivos Específicos

- Realização de testes de *software*;
- Escrever documentação de *software*;
- Implementação de integração contínua;
- Implementaçao do Kivy como *backend* da FGAme;

Estrutura do trabalho

Este trabalho está dividido em cinco capítulos, sendo que o Capítulo 1 refere-se a fundamentação teórica do trabalho, com os conceitos abordados neste trabalho. No capítulo 2 encontram-se detalhes de como a FGAme funciona e sua arquitetura. O Capítulo 4 refere-se a metodologia utilizada no trabalho e como ele foi realizado. No Capítulo 5 encontra-se os resultados obtidos até o momento e por fim, no Capítulo 6 encontram-se as considerações finais sobre este trabalho e também os trabalhos futuros que poderão ser realizados.

1 Fundamentação Teórica

1.1 Corpo rígido e partículas

Partícula e corpo rígido são dois termos da física usados para descrever objetos. É comum encontrar a definição de partícula como um objeto cujas dimensões podem ser desprezadas e corpo rígido como um conjunto de partículas que tem formato fixo. Do ponto de vista matemático, podemos definir partícula como um objeto cujas propriedades angulares não são relevantes, ou seja, sua orientação não é importante no problema abordado. Note que normalmente não conseguimos definir a orientação de um objeto pontual ainda que o conceito de partícula possa ser aplicado a objetos com dimensões não nulas. De forma complementar, podemos definir corpo rígido como um objeto que as propriedades angulares são importantes.

As propriedades cinemáticas comuns tanto para os corpos rígidos e para as partículas são, posição, velocidade e aceleração (BOURG, 2002), também chamadas de propriedades lineares dos objetos. Do ponto de vista dinâmico, também é necessário falarmos sobre a massa e as forças resultantes que atuam nos objetos, falaremos disso nas seção 1.2.

1.1.1 Movimento dos objetos

O movimento dos objetos em um jogo podem ser de translação, rotação ou uma combinação desses dois movimentos.

1.1.1.1 Translação

O movimento de translação pode ser definido intuitivamente como o deslocamento de um corpo em alguma direção, como por exemplo o movimento de um carro em uma rodovia ou então um lançamento de um projétil.

Para (NUSSENZVEIG, 2002), o movimento de translação de um corpo rígido é definido quando quando a direção de qualquer segmento que une duas partículas de um corpo rígido não se altera durante o movimento, ou seja, todas as partículas do corpo rígido possuem a mesma velocidade e aceleração, sendo que para analisar esse movimento basta utilizar qualquer partícula do corpo rígido. Isso é definido como um movimento de translação pura.

Em um sistema de coordenadas cartesianas, podemos definir a posição de um objeto através de suas coordenadas em relação a um dado referencial, podemos descrever

o movimento de uma partícula em um tempo t através das funções $x(t)$ e $y(t)$ (isso em duas dimensões, visto que a FGAme é uma *engine* 2d).

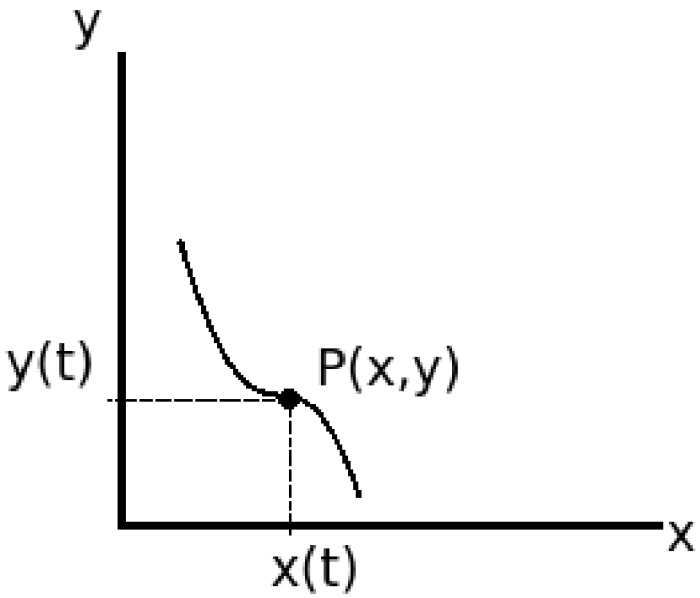

Figura 1 – Movimento de um objeto no plano cartesiano.

O sistema de coordenadas cartesianas nesse caso tem um caráter acessório e o mesmo movimento pode ser descrito utilizando qualquer sistema de coordenadas. É possível descrever a posição ou movimento de um objeto independente do sistema de coordenadas, utilizando vetores (NUSSENZVEIG, 2002). Vetores são objetos matemáticos que são utilizados para representar uma grandeza com magnitude, direção e sentido. No nosso caso, utilizamos o vetor posição $\vec{r}$ como sendo um segmento que liga um ponto no sistema de referência com um ponto no objeto.

A velocidade também pode ser definida como um vetor que possui magnitude e direção, podemos considerar que a velocidade de um carro que trafega a 60km/h, 60km/h é a magnitude da velocidade e a direção é o sentido que o carro toma.

Para calcular a velocidade média de um corpo, tomamos que

$$\vec{v_m} = \frac{\Delta\vec{r}}{\Delta t},$$

onde $\Delta\vec{r}$ é a variação do espaço e Δt é a variação do tempo, isso é bastante útil em alguns casos, mas não é suficiente para descrever um movimento de um corpo. Para descrever o movimento de um corpo, é necessário possuir informação da velocidade instantânea e conseguimos essa informação quando o Δt se torna infinitamente pequeno através de

$$\vec{v} = \lim_{\Delta t \to 0} \frac{\Delta\vec{r}}{\Delta t} = \frac{d\vec{r}}{dt},$$

ou seja, a velocidade instantânea é dado como a derivada do espaço em função do tempo.

Outra propriedade importante dos objetos é a aceleração, que pode ser definida como a mudança na velocidade, temos a aceleração média como

$$\vec{a_m} = \frac{\Delta\vec{v}}{\Delta t}$$

e para a aceleração instantânea temos

$$\vec{a} = \lim_{\Delta t \to 0} \frac{\Delta\vec{v}}{\Delta t} = \frac{d\vec{v}}{dt}$$

e com as informações das velocidades e acelerações instantâneas é possível descrever o movimento dos objetos de uma maneira mais eficiente, sendo que em duas dimensões utilizamos posição, velocidade e aceleração como grandezas vetoriais.

1.1.2 Rotação

O movimento de rotação pode ser definido intuitivamente como girar um objeto em torno de algo, podendo ser um ponto ou girar o objeto em torno de si mesmo, como por exemplo, o movimento de rotação da Terra em torno de si mesmo.

Figura 2 – Rotação da Terra.

O movimento de rotação consiste em fixar um eixo AB em um corpo rígido e qualquer partícula situada nesse eixo deve manter sua distância inalterada de AB e todas as partículas que não estão nesse eixo tem que manter invariável sua distância ao eixo AB de modo que só pode descrever um círculo com centro nesse eixo (NUSSENZVEIG, 2002). Para objetos em um plano, podemos definir o movimento de rotação através de um ponto arbitrário, podendo ser a origem do plano ou um ponto dentro do próprio objeto.

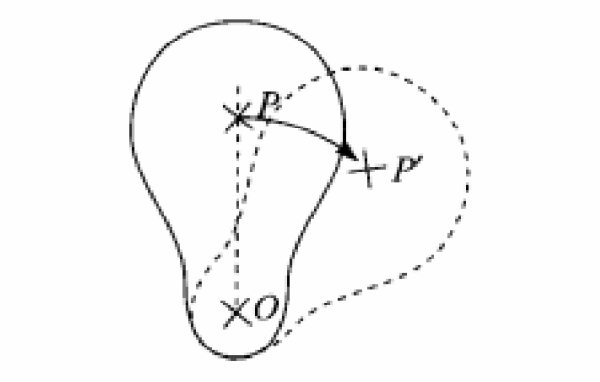

Figura 3 – Rotação de um objeto através de um ponto.

Corpos rígidos são objetos que além das propriedades lineares, possuem propriedades angulares, que são, posição angular, velocidade angular e aceleração angular. Iremos trabalhar somente em duas dimenções, pois em três é bem mais complicado, sendo necessário utilizar matrizes ou *quaternions* para representá-los e foge do escopo deste trabalho, visto que a FGAme é uma *engine* 2d.

A posição angular de um corpo é definida como o ângulo entre uma linha de referência de um objeto e um eixo fixo.

$$\theta = \frac{s}{r},$$

sendo que esse ângulo θ é medido em radianos, s representa o comprimento do arco formado com o eixo x e r é o raio do círculo. A figura abaixo exemplifica a posição angular de um objeto.

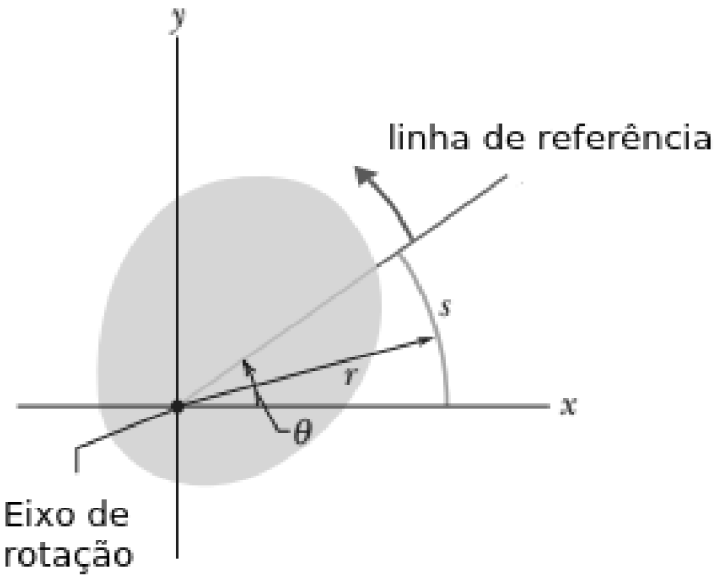

Figura 4 – Posição angular de um objeto.

Definimos velocidade angular como a variação da posição angular, então definimos a velocidade angular média ω_m como a variação da posição angular em um determinado intervalo de tempo, temos

$$\omega_m = \frac{\Delta\theta}{\Delta t}$$

e a velocidade angular instantânea ω quando esse Δt se aproxima de zero de maneira análoga a velocidade linear, temos

$$\omega = \lim_{\Delta t \to 0} \frac{\Delta\theta}{\Delta t} = \frac{d\theta}{dt}.$$

E também de maneira análoga a aceleração linear, temos a aceleração angular como a taxa de variação da velocidade angular sobre o tempo. Então definimos aceleração média α_m como

$$\alpha_m = \frac{\Delta\omega}{\Delta t}$$

e aceleração angular instantânea α como

$$\alpha = \lim_{\Delta t \to 0} \frac{\Delta\omega}{\Delta t} = \frac{d\omega}{dt}.$$

1.2 Cinemática vs Dinâmica

A Cinemática, o ramo da física que consiste em descrever os movimentos, sem se preocupar em explicar as causas. Até o momento, as definições que fizemos de velocidade e acelerações se encontram no campo da cinemática. Em contra partida, determinar as

causas dos movimentos constitui o problema fundamental da dinâmica (NUSSENZVEIG, 2002).

Na física newtoniana, toda alteração de estado de movimento está associada a uma força. Chamamos de força qualquer influência que faz com que um objeto obtenha aceleração e consequentemente altere sua velocidade. A força $\vec{F}$ é uma grandeza vetorial que possui magnitude e direção, caso duas forças ou mais atuem em um corpo, temos a força resultante $\vec{F_r}$, obtemos a força resultante através da aritmética vetorial.

$$\vec{F_r} = \vec{F_1} + \vec{F_2} + ... + \vec{F_N}.$$

Pela experiência do dia a dia sabemos que diferentes corpos respondem de maneiras diferentes quando submetidos a mesma força. A definição física de massa é a característica que relaciona a força aplicada em um corpo e a aceleração resultante. Newton define força resultante em sua segunda lei, como o produto da massa e aceleração,

$$\vec{F} = m\vec{a}.$$

Se a força resultante de um corpo for nula, quer dizer que todas as forças aplicadas nesse corpo se cancelam fazendo com que o objeto se mantenha em movimento retilíneo uniforme. Essas observação constitui o enunciado da primeira Lei de Newton (Lei da inércia). Sendo bastante importante pois caracteriza quais são os observadores inerciais válidos.

Existem vários contextos onde é importante analisar o ponto de aplicação de uma força, quando uma força é aplicada em um corpo que pode ser rotacionado, é produzido uma grandeza vetorial análoga a força chamada de torque, que pode ser definida como

$$\vec{\tau} = \vec{r} \times \vec{F},$$

gerando uma grandeza perpendicular a $\vec{r}$ e $\vec{F}$. Como a FGAme se trata de uma *engine* 2D o torque sempre estará na direção do eixo z, pois $\vec{r}$ e $\vec{F}$ estão no plano xy. A magnitude do torque pode ser obtida através de

$$\tau = rF \sin \phi$$

onde r é a distância do eixo de rotação ao ponto onde a força $\vec{F}$ é aplicada e ϕ é o ângulo de $\vec{F}$ ao eixo de rotação. O torque resultante é a soma de todos os torques em um dado momento, quando o torque resultante é nulo, o objeto não sofre rotação. Caso um corpo não esteja fixo a um eixo de rotação, o mesmo tende a rotacionar em torno do do seu centro de massa.

1.3 Detecção de colisão

Detecção de colisão é um problema de geometria computacional que envolve determinar se dois ou mais objetos se colidiram (BOURG, 2002). Depois que uma colisão é detectada é necessário resolver um problema da física de resposta de colisão.

Como todos os objetos em um jogo podem se colidir, uma simulação com n objetos requer $(n-1)+(n-2)+...+1 = n(n-1)/2$ verificações, o que resulta em uma complexidade de $O(n^2)$, em uma verificação *naive*. Algo que mesmo com um número n de objetos pequeno se torna computacionalmente custoso. Para reduzir esse custo de processamento, é necessário reduzir esse número de verificações, essa redução é feita separando a detecção de colisões em duas fases, uma chamada de *broad phase* e outra chamada de *narrow phase* (ERICSON, 2005).

A *broad phase* identifica pequenos grupos de objetos que possuem maior probabilidade de colisão utilizando caixas de contorno aproximadas como AABB ou círculos e exclui os grupos que com certeza não estão. E a *narrow phase* constitui em testar as possíveis colisões.

1.3.1 *Broad phase*

O algoritmo utilizado na FGAme para detecção de colisão na *broad phase* é o *sweep and Prune*, também conhecido como *Sort and Prune*.

Esse algoritmo consiste basicamente em colocar todos os objetos em uma lista, e a cada *frame* ordenar essa lista através do atributo *xmin* de cada objeto. Então criamos uma lista temporária, e começamos a percorrer a lista de objetos, adicionando o primeiro objeto a lista temporária. Agora comparamos o próximo objeto da lista de objetos com todos os objetos que estão atualmente na lista temporária, caso o atributo *xmin* do próximo objeto for maior que o *xmax* do objeto atual da lista temporária, então remova-o da lista temporária, caso contrário reporte uma possível colisão entre o objeto atual da lista temporária e o objeto atual da lista de objetos e então adicione esse objeto a lista temporária e continue a percorrer a lista de objetos.

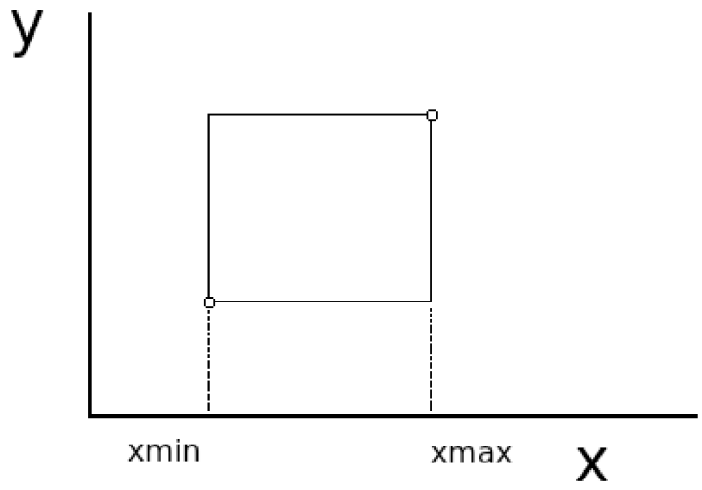

Figura 5 – Atributos xmin e xmax dos objetos da FGAme.

Mesmo com a lista sendo ordenada a cada *frame*, utilizando um bom algoritmo de ordenação de complexidade $O(n \log n)$, conseguimos reduzir a complexidade de $O(n^2)$ para $O(n \log n)$, visto que nessa fase, o problema de detecção de colisão se trata de um problema de busca. É importante ressaltar que na maioria das mudanças de *frame* a posição dos objetos não sofre alteração, então o algoritmo de ordenação opera com complexidade $O(n)$ nesse caso especial, pois a lista já se encontra ordenada.

1.3.2 *Narrow phase*

Os pares reportados como possíveis colisões na *broad phase* são tratados na *narrow phase* com uma verificação entre todos os pares e utilizando algorítimos de detecção de colisão de acordo com os tipos dos objetos tratados, como por exemplo, detecção de colisão entre círculos, retângulos e polígonos.

Existe um algorítimo de colisão genérico que funciona para polígonos convexos *Separating Axis Theorem* (SAT) que diz que "Se dois polígonos convexos não estão sobrepostos, então existe um eixo que passa entre os dois polígonos.". Mas como resolver esse problema em código? Já que existem infinitos eixos em um plano, utilizando os vetores normais únicos de cada figura, no exemplo abaixo, temos dois polígonos A e B, sendo que A está alinhado com os eixos x e y e B não, a partir dos vetores normais únicos de B obtemos dois novos eixos e caso tenha sobreposição das sombras de ambas as figuras em todos os eixos, então há colisão.

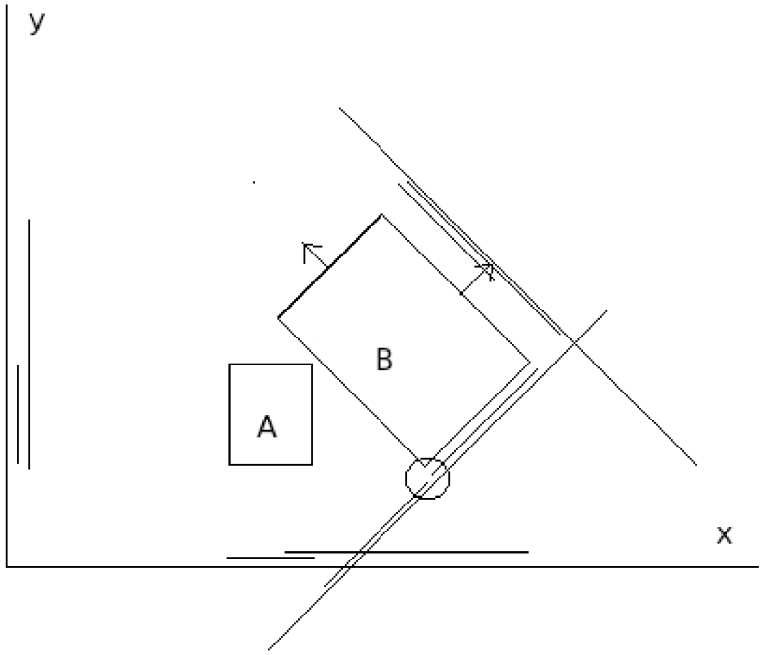

Figura 6 – Dois polígonos sem sobreposição.

O principal objetivo da *narrow phase* é determinar além de que a colisão aconteceu, determinar o ponto onde ocorreu a colisão, algo que pode ser extremamente ambíguo visto que a detecção de colisão só é detectada depois que os objetos estão sobrepostos podendo assim ter vários pontos de colisão e escolher o ponto onde a colisão aconteceu é crucial obter uma resposta de colisão aceitável.

1.4 Resposta de Colisão

Impulso é definido como a força que atua sob um objeto por um curto período de tempo (BOURG, 2002). As forças de colisão que atuam em dois objetos colidindo são forças de impulso, podendo ser impulso linear e impulso angular, que são definidos por

$$\int_{t_-}^{t_+} F dt = m(v_+ - v_-)$$

e

$$\int_{t_-}^{t_+} M dt = I(\omega_+ - \omega_-),$$

respectivamente, onde a massa (m) e o momento de inércia (I) são constantes, sendo que F é a força impulsiva e M é o torque impulsivo, t é o tempo e v é a velocidade. O índice - representa o instante antes do impacto e o índice + representa o instante depois do impacto. Para calcular o impulso médio, podemos utilizar as equações a seguir:

$$F = m(v_+ - v_-)/(t_+ - t_-),$$

$$M = I(\omega_+ - \omega_-)/(t_+ - t_-).$$

Para determinarmos a resposta da colisão, é necessário conhecer a normal de colisão. Conhecendo a normal de colisão, precisamos da velocidade relativa entre os dois objetos, que é a diferença entre os dois vetores velocidade dos dois objetos. Por fim, é necessário calcular a normal relativa a velocidade, que é obtida através do produto escalar entre os vetores de velocidade relativa e normal de colisão.

Energia cinética é a energia associada com os corpos em movimento. A energia cinética é igual a energia requerida para acelerar um corpo do repouso ou para levar um corpo em movimento ao repouso, sendo que existem dois tipos de energia cinética, linear e angular, sendo que energia cinética (K) é uma função da velocidade e massa de um corpo, que podem ser definidas como:

$$K_{linear} = (1/2)mv^2,$$

$$K_{angular} = (1/2)m\omega^2.$$

Conservação de energia cinética entre dois corpos que se colidiram quer dizer que a soma da energia cinética dos dois corpos não se alterou depois do impacto, de acrodo com o princípio da conservação do momento de Newton, que diz que quando dois corpos de massa constante se colidem, o momento é conservado. Isso quer dizer que a soma de suas massas multiplicadas pelas suas respectivas velocidades antes do impacto se mantém depois do impacto,

$$m_1 v_{1-} + m_2 v_{2-} = m_1 v_{1+} + m_2 v_{2+}.$$

Esse tipo de colisão é chamada de colisão perfeitamente elástica. Quando a colisão envolve perda de energia cinética, é chamada de colisão inelástica. Chamamos de colisão perfeitamente inelástica a colisão que quando os corpos colidem eles se juntam e movimentam com a mesma velocidade. Na realidade, esse tipo de colisão é quase improvável, então os impactos são colisões inelásticas. Utilizamos uma variável chamada de coeficiente de restituição (e) que varia entre 0 e 1 e é obtida experimentalmente de acordo com o tipo de colisão, sendo 0 para colisões perfeitamente inelásticas e 1 para colisões perfeitamente elásticas.

Para alterar as velocidades dos objetos e resolver a resposta de colisão, é necessário, aplicar um impulso (J) no ponto de colisão e determinar a velocidade após a colisão.

$$J = m(v_+ - v_-)e = -(v_{1+-v_{2+}})/(v_{1--v_{2-}})$$

Assumindo que o impulso age positivamente no corpo 1 e negativamente no corpo 2 temos:

$$J = m(v_+ - v_-),$$
$$-J = m(v_+ - v_-),$$
$$e = -(v_{1+-v_{2+}})/(v_{1--v_{2-}})$$

Note que temos três variáveis nessas equações, o impulso e as velocidades dos corpos depois do impacto. Como temos três variáveis e três equações, esse sistema possui solução. Conseguimos calcular o impulso através de:

$$J = -(v_{1-} - v_{2-})(e + 1)/(1/m_1 + 1/m_2)$$

Com a normal de colisão (n), conseguimos calcular as velocidades depois do impacto:

$$v_{1+} = v_{1-} + (Jn)/m_1,$$
$$v_{2+} = v_{2-} + (-Jn)/m_2,$$

Caso os objetos sejam círculos, é relativamente simples determinar a normal de colisão, basta pegarmos a normal em direção ao centro de massa do outro objeto. Caso os objetos sejam polígonos, é um pouco mais complicado pois os tipos de contato podem ser, vértice-vértice ou vértice-aresta.

No caso da colisão vértice-aresta, a normal de colisão é sempre perpendicular a aresta envolvida na colisão. No caso vértice-vértice, a normal é ambígua, uma solução seria fazer a normal paralela a linha que conecta os dois centros de massa. O ponto de colisão é o vértice que está envolvido na colisão. De maneira análoga a resposta de colisão entre duas esferas, basta aplicarmos o impulso angular.

1.5 *Figuras* de colisão

Como dito na Seção 1.3, detecção de colisão é um problema de geometria computacional, então para resolver esse problema é necessário implementar em códigos várias *figuras* geométricos para que seja determinado se houve ou não colisão entre dois pares de objetos na *narrow phase*.

A seguir iremos detalhar as principais *figuras* que estão presentes na FGAme.

- AABB:

  Do inglês, *Axis-aligned bounding box*, funcionam como retângulos, mas não permitem rotação. Para representá-las em código, precisamos somente de sua posição e tamanho dos seus lados.

- Círculo:

  Um círculo consiste de um ponto central e infinitos pontos a uma distância r desse ponto. Para representar esse objeto em código, necessitamos somente de um vetor representando o ponto central e um atributo para representar o raio.

- Polígonos:

  Figura mais complexa para detecção de colisão, consiste basicamente de uma série de vértices e sua localização no espaço, possui alguns casos especiais como Retângulos e Triângulos. Atualmente a FGAme só consegue tratar de maneira eficiente as colisões de polígonos convexos.

Vale ressaltar que essas figuras possuem representações tanto na FGAme e na biblioteca smallshapes e que também, figuras importantes como retas, segmentos de retas, raios ainda não estão implementadas.

1.5.1 Colisões entre as *figuras*

- Círculo com Círculo:

  É a detecção de colisão mais simples, basta verificar se a distância entre seus centros é menor que a soma dos seus raios.

- Círculo com AABB:

  É necessário encontrar o ponto na AABB que é o mais próximo do centro do círculo, se a distância desse ponto ao centro do círculo for inferior ao raio do círculo, houve colisão (2D. . . , 2012).

- Círculo com Polígono:

  Para resolver este tipo de colisão é necessário verificar primeiramente se o polígono contém o centro do círculo e depois verificar se a distância do centro do círculo para as arestas do polígono é menor que o raio do círculo se alguma dessas verificações for verdadeira, há colisão, caso contrário, não há colisão entre um círculo e um polígono.

- AABB com AABB:

  Basta utilizar uma versão simplificada do SAT projetando as sombras das AABBs nos eixos X e Y e verificar se houve sobreposição nas sombras.

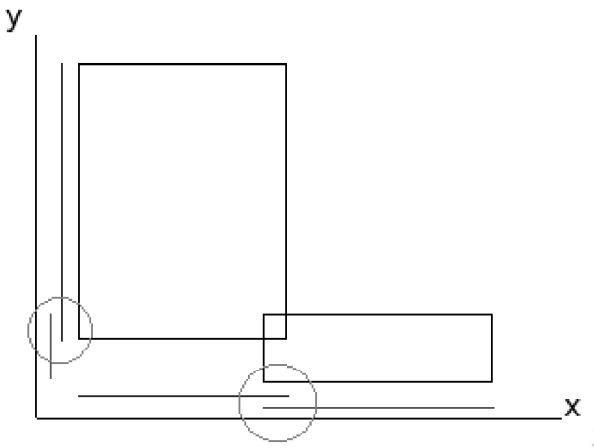

Figura 7 – Sobreposição das sombras de duas AABBs.

- Polígono com Polígono:

  Como dito na Subseção 1.3.2, o algorítmo utilizado para resolver esse tipo de colisão é o SAT. É importante lembrar que esse algorítimo de detecção funciona somente com polígonos convexos e círculos, caso a figura seja um polígono côncavo é necessário adaptar o algorítmo dividino o polígono em polígonos convexos.

- Polígono com AABB:

  Como uma AABB também é um polígono, o mesmo algoritmo de detecção de colisão entre polígonos pode ser usado nesse caso.

2 FGAme

2.1 Dependências

Atualmente a FGAme possui três depêndencias para funcionar, que são as bibliotecas smallshapes, smallvectors e colortools, iremos detalhar essas dependências a seguir. Elas já fizeram parte da FGAme no passado, mas, devido a uma decisão arquitetural, foram separadas em bibliotecas externas para serem utilizadas em outros contextos e facilitar a manutenção das mesmas. A figura 8 mostra a visão geral da arquitetura da FGAme sem entrar em detalhes, basicamente o módulo FGAme se comunica com as dependências e com o módulo *backend*, que se comunica com as dependências e também com a própria FGAme. Backend é a camada mais externa da arquitetura da FGAme e atua na renderização e coleta de *input* do usuário. Atualmente a FGAme suporta como *backend* o Pygame e o Kivy, que está sendo desenvolvido neste trabalho.

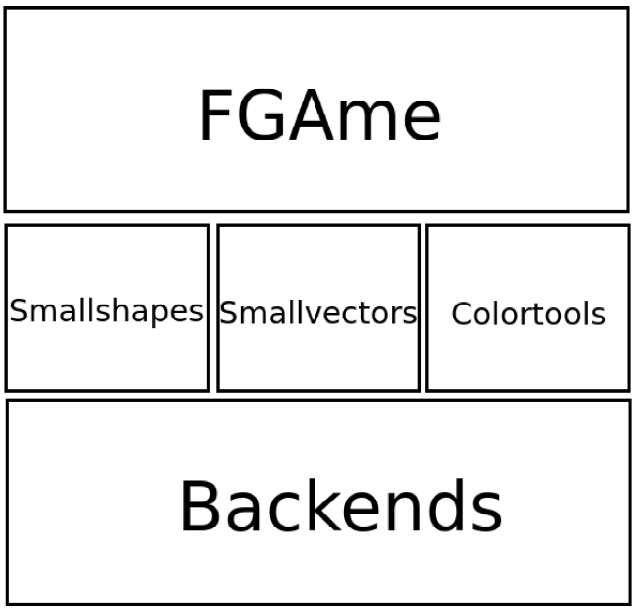

Figura 8 – Visão geral da arquitetura.

2.1.1 smallvectors

A biblioteca smallvectors tem como principal função realizar operações de álgebra linear em objetos de baixa dimensionalidade, como manipulação de vetores, pontos, matrizes e também transformações afins. Para objetos de alta dimensionalidade, a biblioteca Python NumPy [1] pode ser utilizada.

[1] <http://www.numpy.org/>

A arquitetura do smallvectors consiste basicamente em classes abstratas que representam certas propriedades matemáticas, por exemplo, mutabilidade, se podem ser linearizados em vetores de uma dimensão, entre outras. A maioria das classes do smallvectors utilizam de herança múltipla, mas atuam como uma interface e não uma hierarquia propriamente dita. Abaixo desse nível de hierarquia, estão classes de matrizes, vetores e pontos. As classes relacionadas a matrizes, possuem classes como matrizes de rotação, matrizes quadradas e também matrizes arbitrárias. As classes de vetores e pontos são classes relacionadas a vetores de diversas dimensões, vetores unitários e pontos em diversas dimensões. Neste mesmo nível de hierarquia também temos classes responsáveis por realizar transformações afins e de similaridade.

A biblioteca smallvectors está documentada com relação ao uso de vetores, matrizes e transformações afins, mas não possui detalhes na utilização de métodos dessas classes e nem de outras classes que trabalham com Pontos ou Matrizes de rotação.

2.1.2 smallshapes

Biblioteca que visa implementar formas e operações geométricas em 2D e 3D, sendo que a mesma foi criada com a intenção de prover primitivas de colisões para uma *game engine.*

A smallshapes é um pouco mais simples que a smallvectors com relação ao número de classes implementadas, sendo que a smallvectors é uma dependência também da smallshapes. Assim como a smallvectors, a smallshapes possui classes abstratas que ficam em um nível alto de hierarquia que representam propriedades geométricas como por exemplo, se os objetos são convexos, se possuem área, se possuem localização no espaço, etc.

Em um nível mais abaixo de hierarquia, temos algumas classes que representam figuras geométricas, como AABB's(*axis aligned bounding box*, funciona como um retângulo que não permite rotação), diversos tipos de polígonos, como polígonos convexos, polígonos regulares com vários lados, e também casos especiais de polígonos como retângulos e triângulos e por fim, círculos. Também temos classes que implementam elementos primitivos da geometria, como retas, que são infinitas e precisam de dois pontos para serem criadas, segmentos de retas, que funcionam como as retas só que não são infinitas e também raios, que funcionam como retas semi infinitas a partir de um ponto somente.

2.1.3 colortools

Por fim, a última dependência da FGAme é a mais simples e consiste basicamente em uma única classe chamada Color responsável por trabalhar com cores, como por exemplo, definir os valores das cores primárias (vermelho, verde e azul) entre 0 e 255 e também

da transparência de uma cor (*alpha*) entre 0 e 255. Também possui a função de mapear *strings* de cores, como por exemplo a *string 'red'* retornar um objeto Color vermelho, essas cores seguem as especificações do CSS3[2].

A documentação possui informações referentes a instalação da biblioteca e sobre como utilizá-la.

2.2 Arquitetura

Nesta seção iremos detalhar os principais módulos da FGAme, mostrando como os mesmos funcionam em uma visão de alto nível. A Figura 9 mostra de maneira simplificada a hierarquia de módulos da FGAme.

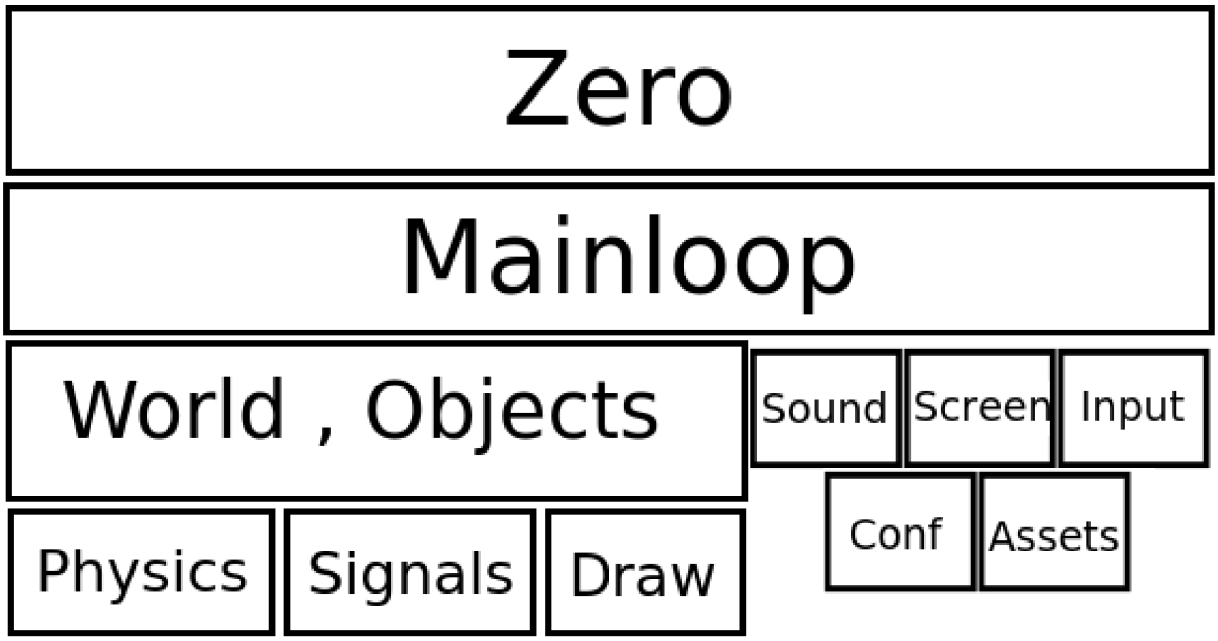

Figura 9 – Arquitetura simplificada da FGAme.

2.2.1 Mainloop

Classe responsável por controlar a posição dos objetos a cada mudança de *frame* através de eventos de *frame enter, frame end* e também pela definição de novos eventos, como por exemplo, agendar eventos de tempo. Também é responsável por chamar os métodos do *backend* responsáveis pela leitura do *input* do usuário e também de atualizar a tela.

[2] <https://www.w3.org/wiki/CSS3/Color/Extended_color_keywords>

2.2.2 World

Módulo responsável por combinar a simulação de física dos objetos com o que está sendo exibido na tela pelo *backend*. É necessário sobrescrever alguns métodos desse módulo em cada *backend*, como os métodos de adicionar um novo objeto no mundo e também o de atualizar a posição de cada objeto através do Mainloop.

2.2.3 Signals

O módulo Signals da FGAme é responsável por definir e disparar eventos e também associar *callbacks* a esses eventos.

A princial classe do módulo é a classe Signal, que tem como principal finalidade representar um sinal. Temos vários sinais já representados na FGAme, como *click* do *mouse*, pressionamento de uma tecla, mudança de *frame*. Também é possível que o usuário crie seus próprios sinais. Para cada sinal, existem *handlers* que são representados pela classe Handler. Esses *handlers* atuam como *callbacks* desses sinais, ou seja, cada vez que um sinal é disparado um *callback* será executado.

2.2.4 Input

Classe abstrata possui vários métodos específicos que são sobrescritos de acordo com o *backend* para resolver cada tipo de sinal identificado, como clique do *mouse* , pressionamento de uma tecla e mudança de *frame*.

2.2.5 Screen

Temos uma classe abstrata Screen, responsável por desenhar os objetos na tela através da sobrescrita dos métodos em cada *backend*, da mesma maneira da classe Input. Temos também uma classe responsável por tratar os eventos relacionados a câmera do jogo, sendo que a mesma funciona como um verificador se o objetos se encontram na tela e fazem uma repintura dos mesmos a cada *frame*.

2.2.6 Draw

Ao contrário do módulo Screen, não tem responsabilidade por desenhar os objetos na tela, mas sim pela representação deles. Associa cores as primitivas da *smallshapes* através da herança múltipla da linguagem Python. Também possui duas classes responsáveis por representar texturas e imagens, mas que ainda estão em beta.

2.2.7 Backends

Módulo responsável por coordenar os módulos de Mainloop, Input e Screen de acordo com o *backend* selecionado. É necessário a criação de dois arquivos, um contendo as classes sobrescritas utilizando as funções de cada *backend* e outro contendo informações de configuração. Atualmente a FGAme oferece suporte ao Pygame e PyGameGFX somente, sendo que o Kivy está sendo desenvolvido neste trabalho.

Existem vários arquivos criados para oferece suporte futuramente a diversos *backends* como Qt, SDL2, PyGameGL entre outros. Temos um *backend* de teste, onde o mesmo não abre nenhum tipo de telae e é utilizado para realização de testes unitários.

2.2.8 Physics

Esse é o maior módulo da FGAme, que é dividido em sub módulos.

2.2.8.1 Bodies

Módulo dentro do módulo Physics, responsável por instanciar os objetos que são utilizados na FGAme. Sendo que possui uma classe genérica chamada Body que é responsável por armazenar as propriedades físicas dos objetos, como orientação, massa, momento de inércia,velocidade, densidade, etc. Essa classe também é responsável por algumas propriedades geométricas, como uma caixa de contorno que envolve o objeto (AABB), área, raio de giração e algumas caixas de contorno simplificadas. Também possui vários métodos relacionados a física, como aplicação de forças nesse objeto, rotacionar objetos, aplicar torque e aceleração angular.

Nesse módulo temos classes que herdam de Body que lidam com as peculiaridades de cada tipo de objeto como AABB's, polígonos e círculos. Nessas classes específicas, estão definidas os métodos de detecção de colisão, de acordo com os tipos dos objetos, como por exemplo, detecção de colisão de círculo com AABB. Essa especificação é definida através da biblioteca Pygeneric, visto que o recurso de *multiple dispatch* não tem suporte nativo na linguagem Python.

2.2.8.2 Detecção de Colisão

É no módulo *broadphase* onde estão implementados os algorítimos de detecção de colisão da FGAme, sendo que existem três abordagens desses algorítimos, utilizando AABB's , CBB's que envolvem os objetos e também uma abordagem mista, no qual utiliza as AABB's e CBB's dos objetos simultaneamente. Por padrão é utilizado a abordagem mista, mas o usuário pode escolher qual abordagem utilizar. Essa fase retorna uma lista de possíveis colisões, que é tratada na *narrowphase* onde serão utilizados algorítimos de detecção de colisão específicios de cada tipo de objeto envolvido no problema.

2.2.8.3 Colision

O módulo *colision* é responsável por declarar um objeto do tipo Colision que possui as informações e métodos necessários para resolução de uma colisão que foi detectada na *narrowphase*, como por exemplo, remover a sobreposição dos objetos, resolver a colisão dos objetos, aplicar um impulso nos objetos.

2.2.8.4 Simulation

Módulo responsável pela simulação da física da FGAme, sem qualquer ligação com o *backend*. Objetos de Simulation também definem constantes globais como gravidade, atrito, amortecimento e coeficiênte de restitiuição e que podem ser sobreescritos por objetos específicos e que esses objetos definem sua própria gravidade, atrito, etc, sendo que a simulação de física funciona de acordo com os valores desses atributos. A classe Simulation é responsável por atualizar a física do sistema e calcular todas as interações dos diferentes objetos que aparecem na cena.

O principal método da classe é o método *update*, que recebe um intervalo de tempo como argumento e utiliza-o para incrementar a simulação e que é chamado pelo Mainloop a cada mudança de *frame*, onde visa atualizar as acelerações, velocidades e posições, e também detectar as possíveis colisões da *broadphase* e através das possíveis colisões retornadas na *broadphase*, resolver as colisões reais da *narrowphase*.

2.2.9 Objects

Como a arquitetura da FGAme é feita em camadas, é importante isolar a parte da simulação da física da visualização para um não dependa do outro. Este módulo é responsável por unir o módulo physics.Body e o módulo Draw, sendo que o mesmo possui uma classe chamada Body que herda de physics.Body que possui alguns atributos relacionados a imagem do objeto, cor do objeto, cor e espessura da linha do objeto e se o mesmo é visível ou não. Também possui métodos que alteram esses atributos.

Possui classes específicas de AABBs, polígonos e círculos que herdam da respectiva classe do módulo physics e também de Body, no qual essas classes possuem métodos de desenhar na tela nos *backends* que utilizam *canvas*.

2.2.10 Configuration

Módulo responsável por instanciar um objeto da classe Configuration global, através de um Singleton. Esse objeto é responsável por inicializar o *backend*, *screen*, *input* e *mainloop* de maneira padrão. Através da instância desse objeto, é possível alterar os valores padrão de cor do *background*, resolução, *framerate*, duração do *frame* e também *backend* utilizado.

2.2.11 Assets

Módulo responsável por gerenciar os recursos multimídia da FGAme, sendo que esses recursos podem ser imagens ou arquivos de áudio. Para funcionar corretamente é necessário organizá-los nos diretórios específicos para cada um dos tipos de recurso multimídia.

2.2.12 Sound

Módulo responsável por gerenciar os sons da FGAme, que podem ser do tipo *SFX* ou *Music*, sendo que os dois herdam da classe Sound que possui métodos de iniciar, parar, pausar e resumir um som, sendo que os formatos aceitos são *wav, ogg* e*mp3*.

A diferença entre música e *sfx* é que é possível instanciar somente um objeto do tipo Music e vários objetos do tipo SFX. A classe Music sobrescreve os métodos de Sound para que seja garantido que somente uma música esteja ativa, enquanto a classe SFX possui métodos de inciar, parar, pausar e resumir vários sons ao mesmo tempo.

2.2.13 Patch

As funções presentes neste módulo permitem modificar o comportamento da FGAme de maneira arbitrária. O propósito deste módulo é de caráter didático, fazendo com que os estudantes que estejam utilizando a FGAme consigam re-implementar partes específicas da FGAme para fixar alguns conceitos de física para jogos. Este módulo utiliza o recurso de *monkey patching* que a linguagem Python oferece, que permite realizar mudanças em classes e métodos em tempo de execução.

2.2.14 Zero

Módulo responsável por reduzir o *boilerplate* na FGAme, o termo *boilerplate* na engenharia da computação refere a seções de código que são incluídas em vários locais do projeto com pouca ou nenhuma alteração. Esse módulo basicamente consiste em prover uma infraestrutura básica para o usuário leigo consiga criar objetos da FGAme imediatamente. O módulo zero provê uma instância de World e Conf com todos os sinais da FGAme já registrados. O exemplo abaixo mostra como é simples a criação de um Pong utilizando a FGAme.

```
from FGAme import *

world.add.margin(10)

ball = world.add.circle(20, pos=pos.middle, color='red')
ball.vel = vel.random()
```

```
p1 = world.add.aabb(shape=(20, 120), pos=(30, 300), mass='inf')
p2 = world.add.aabb(shape=(20, 120), pos=(750, 300), mass='inf')

on('long-press', 'w').do(p1.move, 0, 5)
on('long-press', 's').do(p1.move, 0, -5)
on('long-press', 'up').do(p2.move, 0, 5)
on('long-press', 'down').do(p2.move, 0, -5)

run()
```

2.2.15 Demos

Esse módulo não visa implementar nenhuma funcionalidade para a FGAme, mas sim armazenar alguns exemplos em código de como a FGAme funciona e também testar algumas funcionalidades como colisões e desenhos de objetos na tela.

Possui um sub módulo chamado de games onde estão exemplos de jogos simples que foram feitos utilizando a FGAme como o clássico Pong e uma versão do famoso jogo Flappy Bird. Também possui um módulo de simulações, onde são implementados algumas simulações de física, como moléculas de gás se chocando, pêndulo de Newton, pistão de um motor e empilhamento de objetos.

2.3 Backends

Esta seção visa mostrar quais os *backends* que a FGAme utiliza atualmente, que são o Pygame e o Kivy, sendo que o Kivy está sendo introduzido na FGAme devido a este trabalho.

2.3.1 Pygame

Pygame é uma biblioteca escrita em Python para criação de aplicações multimídia como jogos utilizando como base a biblioteca SDL (PYGAME, 2017). SDL é uma biblioteca que provê acesso ao audio, teclado, mouse e *hardware* de gráfico, sendo bastante utilizada em emuladores, softwares de vídeo e jogos (SDL, 2017).

O Pygame atua como API da SDL em Python, já que a SDL é escrita em C.

2.3.2 Kivy

Kivy é um *framework* escrito em Python de código aberto utilizado para o desenvolvimento de aplicações que contenham interfaces de usuário (KIVY, 2017). Com o Kivy é possível fazer uma aplicação rodar nas plataformas Android, Linux, Windows, OS X e iOS com o mesmo código Python e utilizar vários recursos nativos dessas plataformas.

Como o Kivy está sendo *backend* Kivy está sendo desenvolvido neste trabalho, iremos entrar em mais detalhes de como funciona sua arquitetura.

2.3.2.1 Arquitetura do Kivy

A arquitetura[3] do Kivy consiste em vários blocos que podem ser agrupados de acordo com o nível de abstração como na Figura 10. Iremos explicar de maneira breve os blocos que utilizamos neste trabalho.

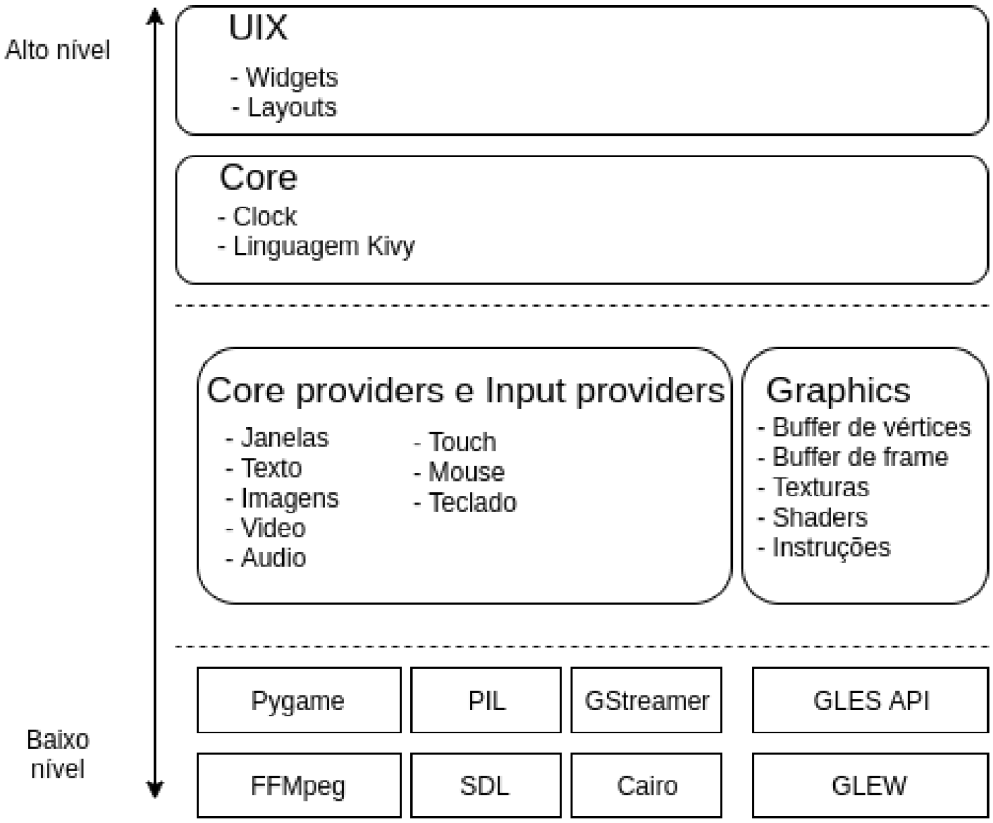

Figura 10 – Arquitetura simplificada do Kivy

- UIX

  O módulo UIX do Kivy se refere a criação de interfaces de usuário, que contém *widgets* e *layouts*. Widgets se referem aos elementos da interface que adicionam a aplicação um tipo de funcionalidade, como por exemplo botões e listas. Layouts se referem a uma maneira de organizar esses *widgets*.

  Um dos objetivos deste trabalho foi criar um *widget* chamado de FGAmeWidget que cria uma instância da FGAme dentro de uma aplicação Kivy.

- Core

[3] <https://kivy.org/docs/guide/architecture.html>

Na camada Core existem classes responsáveis por controle de eventos e para manter o *framerate* da aplicação. O Kivy também possui sua própria linguagem para facilitar o desenho de objetos na tela, tirando do código Python as especificações de cada tipo do Kivy, como por exemplo cor e tamanho de um círculo, isso permite substituir código Python por uma linguagem mais declarativa. O lado direito da Figura 11 exemplifica o uso da linguagem Kivy em conjunto com código Python e no lado esquerdo a mesma aplicação utilizando somente código Python. Apesar da linguagem Kivy ser muito útil para criar interfaces gráficas, ela não foi utilizada na FGAme.

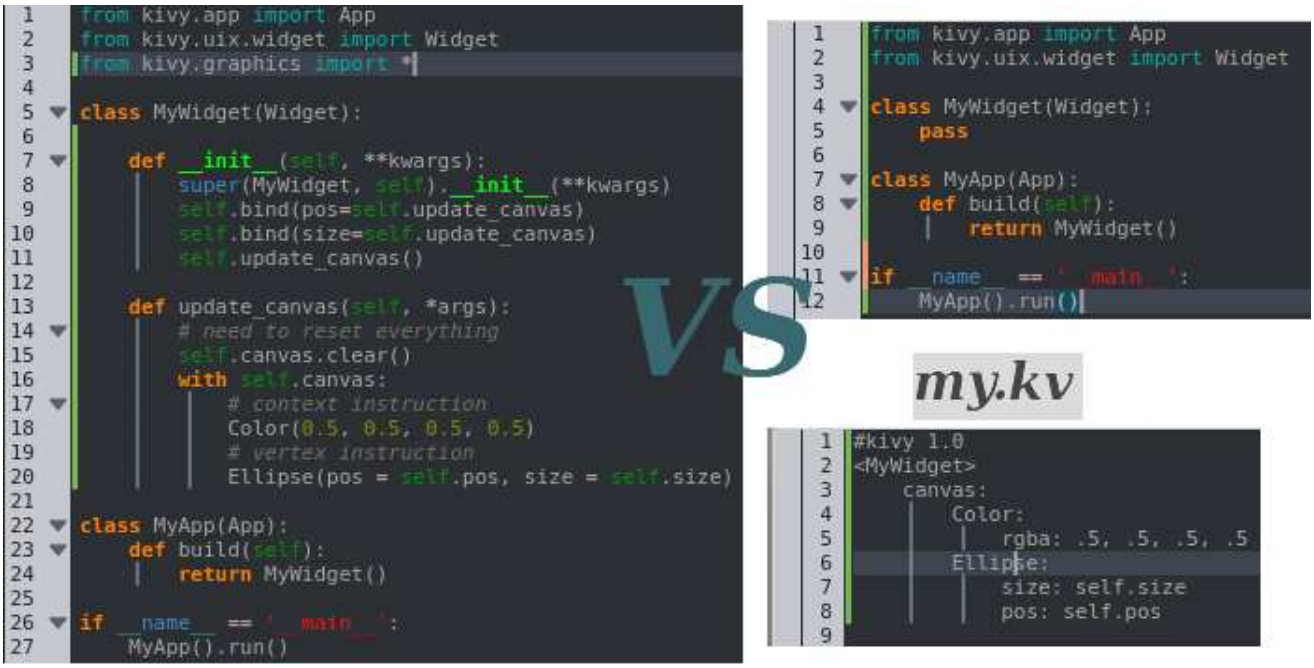

Figura 11 – Uso da linguagem do Kivy.

- *Core providers* e *Input providers*:

  O Kivy abstrai tarefas como abrir uma janela, mostrar imagens e reproduzir sons, etc e chamam-as de tarefas *core*. Por exemplo, dependendo do sistema operacional existem diferentes APIs para essas tarefas e então os *core providers* atuam como uma camada de comunicação entre a aplicação Kivy e os diferentes sistemas operacionais. O mesmo se aplica aos *input providers* que atuam como uma camada de comunicação entre os eventos de *input* como pressionar as teclas, cliques do *mouse* e *touch* na tela.

- Graphics

  A API gráfica do Kivy é uma abstração da OpenGL e é escrita em C/Cython por questões de performance. No nível mais baixo, o Kivy utiliza comandos da OpenGL

para renderizar os objetos na tela, sendo que escrever código OpenGL pode ser um pouco complicado, especialmente para iniciantes, então a API gráfica do Kivy facilita a vida do usuário deixando-o desenhar objetos na tela utilizando comandos simples.

3 Práticas da Engenharia de Software na FGAme

Este capítulo exibe algumas práticas da Engenharia de Software que foram utilizadas na FGAme. A Engenharia de Software pode ser definida como uma disciplina que aplica os princípios da engenharia tradicional para a produção de software de qualidade (RABIN, 2010).

3.1 Testes automatizados

O principal objetivo do teste de *software* é executar o código e detectar a presença de *bugs*. A probabilidade de identificar *bugs* no código depende da eficiência da suíte de testes, quanto mais caminhos são testados, maior a probabilidade de se detectar *bugs* no código.

Normalmente, a atividade de teste de *software* é desafiante e tediosa especialmente quando a complexidade e o tamanho do programa são grandes (BOOPATHI et al., 2017). Mesmo a FGAme sendo uma *game engine* relativamente simples, durante o início do seu desenvolvimento não foi dado a devida atenção a implementação de testes fazendo com que escrita de novos testes para a FGAme seja uma tarefa bastante custosa, principalmente os métodos que trabalham com a renderização de imagens, emissão de sons e também os métodos de interação do usuário com a *engine*.

Atualmente a FGAme possui uma cobertura de testes atualmente de 55% utilizando a ferramenta Coverage.py [1]. A cobertura de testes do smallvectors atualmente se encontra em 81% e está com *build* funcionando nas versões 3.4 em diante do Python na ferramente de Integração contínua Travis CI. A *build* da smallshapes está funcionando em todas as versões do Python e conta com uma cobertura de 49%. Assim como a smallshapes, a *build* da colortools está funcionando em todas as versões do Python e conta com uma cobertura de testes de 100%.

3.2 Documentação

Todos os projetos de Software independentemente da aplicação, geram documentação de *software* associada (SOMMERVILLEN, 2001). Um *software* que não possui boa documentação pode fazer com que os usuários do *software* cometam erros e os desenvolvedores que irão trabalhar na manutenção do *software* entendam funcionalidades do

[1] <https://coverage.readthedocs.io/en/coverage-4.5.1/>

software de maneira errônea e consequentemente não escrevam códigos que funcionem de maneira apropriada.

(SMITH, 2001) diz que desenvolvedores responsáveis por dar uma manutenção em um programa, gastam aproximadamente metade do tempo simplesmente tentando entender a função do código fonte e que a manutenção e evolução de *software* consome mais de 80% do ciclo de vida de um *software* (VAGAč; KOLLáR, 2012), ou seja, a documentação de *software* está diretamente associada a curva de aprendizado do código fonte, *software* com boa documentação tende a ter uma curva de aprendizado menor. Levando esses dados em consideração, fica claro que a documentação de *software* é uma prática bastante importante da Engenharia de Software.

Para (SOMMERVILLEN, 2001), existem dois tipos de documentação de *software*, Documentação de Processo, que visa registrar o processo de desenvolvimento e manutenção e Documentação de Produto, que visa descrever o produto do ponto de vista do desenvolvedor responsável por desenvolver e dar manutenção no sistema e do ponto de vista do usuário que visa descrever como o sistema funciona e como utilizá-lo. Atualmente a FGAme possui Documentação de Produto, visto que no caso de uma *game engine* os papéis de usuário e desenvolvedor se fundem, pois uma *engine* atua na criação de outros jogos então um usuário também é um desenvolvedor.

Na FGAme, existe uma documentação com as instruções de como instalar no sistema operacional Linux, mas não no Windows e Mac OS X. Com relação ao uso do *software*, existe uma documentação parcial, com alguns exemplos de como utilizá-la e referência para alguns métodos, mas nada tão detalhado que uma pessoa consiga utilizá-la sem olhar o código fonte. E com relação ao código fonte, alguns métodos e classes estão documentos em português devido a FGAme ter sido criada com o código todo em português e depois ela foi migrada para o inglês. Os trechos de códigos mais antigos e que não foram refatorados ainda estão em português e a tendencia é que o código fique totalmente em inglês no futuro.

3.3 Empacotamento

Gerência de configuração de Software é um conjunto de atividades que foram projetadas para gerenciar mudanças identificando os produtos que estão disposto a mudar (RABIN, 2010). É uma área da Engenharia de Software que visa fornecer apoio ao desenvolvimento de *software*. Como o próprio *software* muda constantemente é necessário ferramentas que visam facilitar o gerenciamento dessas mudanças, como por exemplo um gerenciador de pacotes que é uma ferramenta que oferece um meio automático de instalar, atualizar e remover pacotes de *software* em um sistema operacional.

A FGAme, smallshapes e smallvectors estão empacotadas no repositório oficial[2] de pacotes da linguagem Python, sendo que todos os pacotes são instalados através do gerenciador de pacotes pip [3]. Sendo que através do pip é possível instalar e remover pacotes Python sem a necessidade de privilégios de segurança de outro usuário do sistema operacional. É recomendável instalar as versões empacotadas do sistema operacional do Kivy e do Pygame, visto que eles necessitam da SDL e existem problemas com relação ao empacotamento de bibliotecas que possuem módulos e bibliotecas externas da linguagem C, esse é um problema conhecido na comunidade Python.

3.4 Integração contínua

Integração contínua é uma prática de desenvolvimento que surgiu da metodologia ágil de desenvolvimento Extreme Programming, que visa integrar o desenvolvimento do time continuamente. A integração contínua, de maneira abstrata, funciona da seguinte maneira: quando um desenvolvedor faz uma alteração no repositório do projeto, um sistema automático verifica o código, roda um conjunto de comandos e verifica se aquela alteração não introduziu erros ao programa através da geração de uma *build* de *software* (MEYER, 2014). Essa *build* depende do tipo do *software* que está sendo desenvolvido, caso o *software* seja compilado, essa *build* é a compilação bem sucedida do *software* e também a verificação do *software* através de uma série de testes automatizados. Caso o *software* seja interpretado, que é o caso da linguagem Python, a *build* é gerada através da verificação do *software* através de testes automatizados.

A FGAme utiliza o TravisCI[4] como serviço de Integração Contínua. Como a FGAme é um *software* de código aberto, o serviço funciona de maneira gratuita. O TravisCI primeiramente requer informações de como gerar a*build* do *software*. Após essa etapa inicial, no caso da FGAme, é necessário rodar um *script*, onde os testes automatizados serão executados em várias versões do Python e por fim, caso os testes não tenham falhado é necessário rodar um *script* de sucesso. Novamente, no caso da FGAme, é atualizar a cobertura de testes na plataforma Coveralls[5], mas poderia ser outros *scripts*, como por exemplo, atualizar a versão da FGAme no pip. Atualmente existe um problema com a cobertura de testes apresentada pela plataforma Coveralls, apresentando uma cobertura de 33 %, sendo que a cobertura real apresentada pelo Coverage.py é de 55%.

[2] <https://pypi.python.org/pypi>
[3] <https://pip.pypa.io/en/stable/>
[4] <https://travis-ci.org/>
[5] <https://coveralls.io/>

4 Metodologia

Este capítulo mostra como foi desenvolvido este trabalho, descrevendo o planejamento do mesmo, metodologia e ferramentas utilizadas durante sua execução.

4.1 Planejamento

Foi feito um planejamento inicial de acordo com o tempo disponível para realização deste trabalho, com um total de 14 semanas, contendo atividades relacionadas a elaboração do texto deste trabalho e também de implementação. As atividades foram estruturadas de acordo com a estrutura do trabalho escrito com a adição de atividades de implementação sendo que esse planejamento inicial foi elaborado somente como referência do andamento do trabalho.

4.2 Kanban

Como somente uma pessoa participou do desenvolvimento desse trabalho, não faz sentido utilizar uma metododologia formal de desenvolvimento de *software* como Scrum e XP. Então utilizamos o Kanban como uma técnica de auxílio, devido a mesma possuir uma boa visualização do fluxo de trabalho e consequentemente mostrar o andamento do trabalho (LEI et al., 2008).

O Kanban consiste basicamente de um quadro que possui um fluxo de trabalho que é organizado em colunas. Um Kanban simples possui três colunas normalmente, *To Do, In Progress* e *Done*, sendo que esse quadro pode ser modificado de acordo com as necessidades do time (RADIGAN,). Foram utilizadas as colunas citadas anteriormente com a adição da coluna de *Backlog*, onde todas as atividades do planejamento inicial estiveram inicialmente e estão marcadas como pendentes. A coluna *To Do* representa as próximas atividades a serem realizadas, a coluna *In Progress* representa as atividades que estão sendo realizadas no momento e a coluna *Done* representa as atividades que já foram concluídas. Utilizamos a aplicação Trello[1] como Kanban.

[1] <https://trello.com/>

Tabela 1 – Planejamento inicial.

Semana	Texto	Implementação
1	Dependências	
2	Arquitetura FGAme - Signals, World, Mainloop, Input - Screen, Draw, Backends	
3	Fundamentação Teórica - Shapes de Colisão - Detecção de Colisão	Testes de colisão
4	Fundamentação Teórica - Rotação e Translação - Corpos rígidos e Partículas - Cinemática e Dinâmica	
5	Arquitetura FGAme - Physics, Objects, Configuration	Testes de física
6	Arquitetura FGAme - Actions, Assets, Demos - Sounds, Patch, Zero	
7	Arquitetura FGAme - Backends	Integração Kivy como Backend e Testes
8	Introdução	
9	Práticas Engenharia de Software - Cobertura de testes, documentação	Documentação Kivy-Backend
10	Práticas Engenharia de Software - Empacotamento, Integração Contínua	Implementação Integração contínua
11	Metodologia	
12	Resultados	Deploy para Android
13	Resultados	Aplicativo de carregar Jogos
14	Conclusão	Implementação do que está faltando

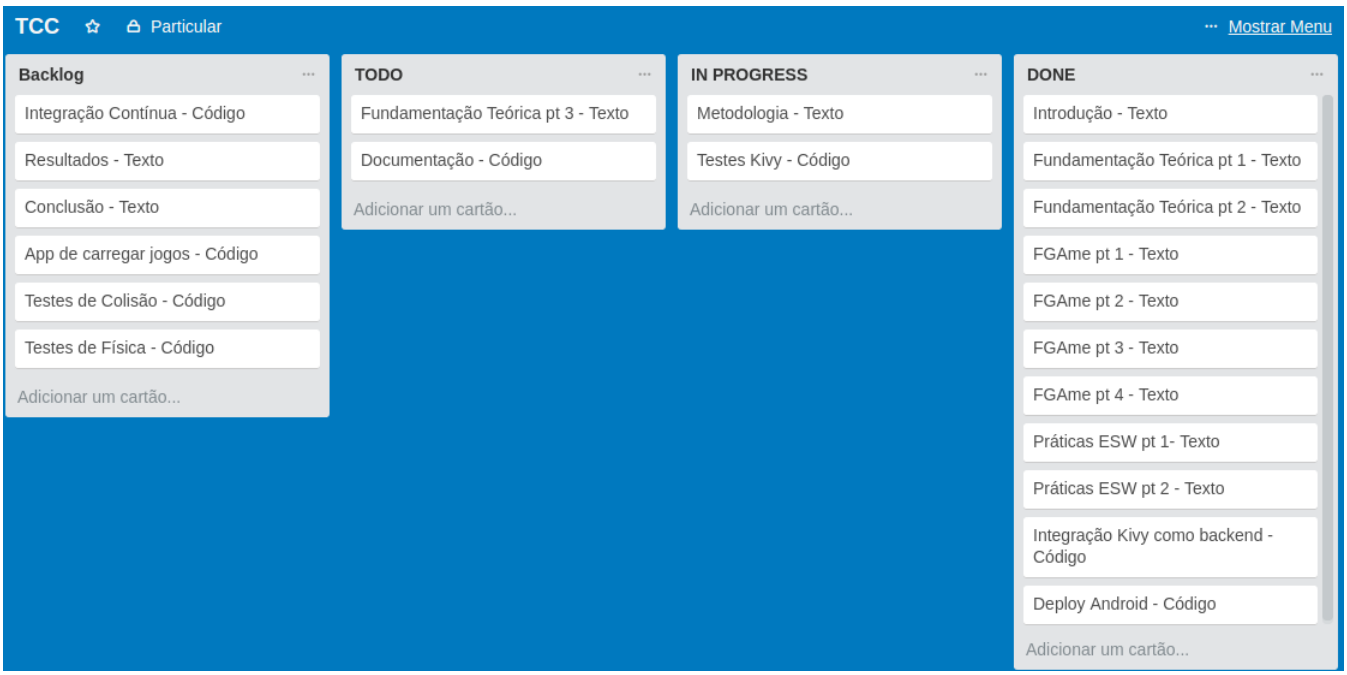

Figura 12 – Kanban mostrando o andamento do projeto.

4.3 Ferramentas

4.3.1 Ambiente de desenvolvimento

O sistema operacional utilizado foi o Ubuntu em sua versão 16.04 e a linguagem de programação utilizada foi o Python na versão 3.5 que é a versão que acompanha o Ubuntu 16.04.

O hardware utilizado foi um Intel Core i7-5500U 2.4 GHz com 8GB de memória RAM e o celular Android utilizado para testes foi um Motorola Moto G de terceira geração, 1GB de memória RAM e rodando a versão 6.0 do sistema operacional Android.

4.3.2 Bibliotecas

4.3.2.1 Buildozer

Para geração do APK Android, foi utilizado a biblioteca Python Buildozer [2], que é uma biblioteca do próprio Kivy para transformar uma aplicação Kivy em uma aplicação para Android e para iOS, a versão utilizada do Buildozer foi a 0.34dev. Essa biblioteca oferece suporte para Python 2 e também para Python 3, sendo que é necessário utilizar uma versão de testes para gerar o APK de uma aplicação que utiliza o Python 3.

4.3.2.2 Cython

Cython é uma linguagem de programação que faz com que seja possível escrever extensões da linguagem C para a linguagem Python de maneira simples. Ela visa ser um conjunto da linguagem Python que oferece programação de alto nível, programação orientada a objetos e funcional. Além de oferecer declaração estáticas de tipo fazendo com que o código fonte Cython gere código C/C++ otimizado e compilado como extensões Python e isso permite com que a execução seja muito mais rápida e permita integração com bibliotecas externas da linguagem C mantendo a alta produtividade da linguagem Python (CYTHON, 2018).

O Buildozer e o Kivy dependem do Cython, para a geração do APK Android, foi necessário utilizar a versão utilizada 0.23 do Cython. Utilizamos a versão 1.9.2-dev0 do Kivy.

4.3.2.3 Pytest

Para realização de testes de *software*, foi utilizado o *framework* de testes Pytest na versão 3.0.7 visto que a FGAme já utilizava este *framework*.

[2] <https://github.com/kivy/buildozer>

4.3.3 Docker

O Docker é uma ferramenta utilizada para criação, *deploy* e utilizações de aplicações utilizando *containers* (OPENSOURCE.COM, 2018). A diferença de um *container* Docker para uma máquina virtual é que o *container* utiliza o mesmo *kernel* do sistema operacional em que o Docker está instalado, enquanto em uma máquina virtual isso não acontece. Ou seja, as dependências que já estão no sistema operacional em que o Docker está rodando são reaproveitadas caso elas sejam necessárias na aplicação dentro do *container*, ganhando *performance* em relação as máquinas virtuais.

Outro ponto positivo do Docker é a facilidade de distribuição de uma Imagem Docker. Uma vez funcionando, basta subir a Imagem para o DockerHub[3] e então todos os usuários que tenham o Docker instalado conseguirão criar os *containers* a partir daquela imagem e conseguir rodar a aplicação desejada.

4.4 Alterações no planejamento

Algumas alterações do escopo do trabalho foram feitas durante o seu desenvolvimento como:

- Remoção do escopo a implementação do aplicativo de carregar jogos.
- Criação de uma imagem Docker para geração do APK Android.
- Seção de resposta de colisão.

[3] <https://hub.docker.com/>

5 Resultados

Este capítulo visa detalhar os resultados obtidos com relação aos objetivos propostos deste trabalho.

5.1 Implementação do *backend* Kivy

Primeiramente integramos a FGAme ao Kivy sem se preocuparmos com a arquitetura de *backends* da FGAme utilizando uma classe chamada KivyWorld como proposta de solução e posteriormente integramos essa solução aos *backends* da FGAme. Posteriormente, refatoramos a primeira solução, pois a mesma não era bem otimizada, usando a arquitetura de sinais da FGAme.

5.1.1 Solução KivyWorld

5.1.1.1 Integração do sistema gráfico do Kivy com a FGAme

Para conseguir renderizar um objeto da FGAme utilizando o Kivy foi, necessário criar uma nova classe chamada de KivyWorld que herda de World da FGAme e que possui um *widget* do Kivy que será utilizado para adicionar os objetos na tela.

```
class KivyWorld(World):

    def __init__(self, world):
        super().__init__()
        self.gravity = world.gravity
        self.objects = []
        self.widget = FGAmeWidget(self)
        self.app = FGAmeApp(self, self.widget)
        self.is_paused = True
        for obj in world._objects:
            self.add(obj)

    def _add(self, obj, layer=0):
        super()._add(obj, layer)
        if isinstance(obj, Body):
            self._simulation.add(obj)
            obj.world = self
        register_to_canvas(obj, self)
```

Toda vez que um objeto é adicionado no mundo da FGAme é chamado o método privado *_add* que adiciona um objeto na simulação. Sobrescrevemos esse método, adi-

cionando uma chamada a função *register_to_canvas* que através da biblioteca Python *singledispatch*, consegue declarar funções com diferentes implementações de acordo com o tipo do objeto passado no primeiro argumento. O exemplo abaixo mostra como um Círculo é desenhado no *canvas* do Kivy.

```
@singledispatch
def register_to_canvas(obj, world):
    raise TypeError()

@register_to_canvas.register(Circle)
def _(obj, world):
    with world.widget.canvas: # Adds a colored circle to canvas object
        diameter = 2 * obj.radius
        Color(*obj.color.rgbf)
        kcircle = Ellipse(pos=obj.pos_sw, size=(diameter,diameter))
        world.objects.append(KivyObjectWrapper(obj, kcircle))
```

A classe KivyWorld possui uma lista de objetos que armazenam objetos do tipo KivyObjectWrapper, que contém as posições dos objetos da FGAme e do Kivy, além do valor da posição inicial e também os valores das translações e rotações que serão realizadas.

```
class KivyObjectWrapper:
    def __init__(self, obj_fgame, obj_kivy, translation=None, rotation=
                                            None):
        self.obj_fgame = obj_fgame
        self.obj_kivy = obj_kivy
        self.translation = translation
        self.rotation = rotation
        self.initial_pos = obj_fgame.pos
```

Foi necessário sobrescrever o método *update*, que era responsável por atualizar as posições dos objetos da FGAme e que agora, percorre a lista de objetos do tipo KivyObjectWrapper e atualiza as posições dos objetos no *canvas* do Kivy. Caso o objeto tenha propriedades angulares, é necessário atualizar a sua posição pela modificação dos atributos de translação e rotação, caso contrário, basta atribuir a posição do objeto Kivy com o valor da posição do objeto da FGAme.

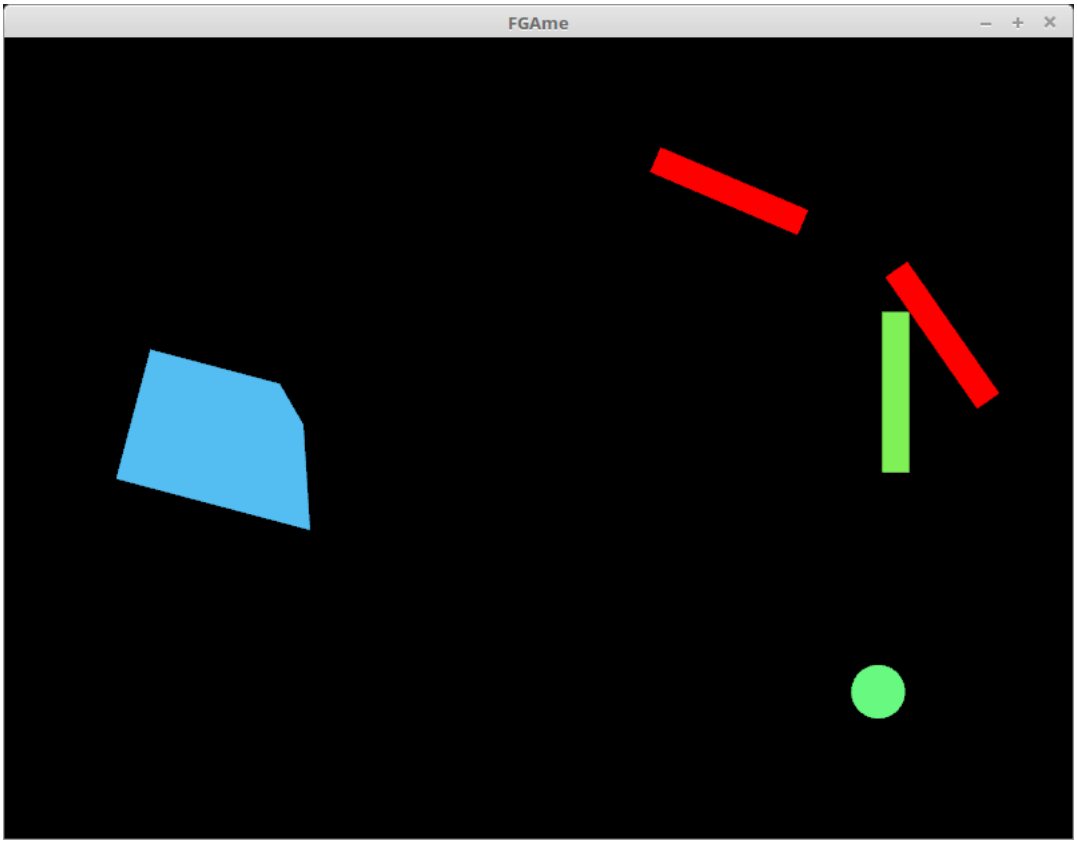

Figura 13 – Aplicação Kivy com a FGAme controlando a física.

5.1.1.2 Integração do I/O da FGAme com o do Kivy

A FGAme foi projetada para que seja possível adicionar um novo *backend* de maneira simples. Através da herança das classes Input, Mainloop e Canvas, que são responsáveis pela interação do usuário com a FGAme, controlar as mudanças a cada *frame* e desenhar os objetos na tela, respectivamente. No arquivo *kivy_conf*, é passado os nomes das classes que herdam de Input, Mainloop e Canvas e também o *import* necessário para utilizar esse *backend*.

```
mainloop = 'KivyMainLoop'
screen = 'KivyCanvas'
input = 'KivyInput'
imports = ['kivy']
```

Na classe KivyInput, registramos todos os eventos do Kivy, como um toque na tela ou pressionar uma tecla e toda vez que um evento desse tipo acontece, é colocado um objeto do tipo Event em uma fila, que será processado pela sobrescrita do método *poll*, que é responsável por escutar todos os eventos que acontecem.

5.1.1.3 Configuração do *backend* Kivy

Para que a FGAme aceite o Kivy como *backend* padrão, foi necessário adicionar a *string 'kivy'* na lista de *backends* que a FGAme suporta na classe Configuration, para que seja possível o usuário selecionar o Kivy como *backend*, já que o padrão é o Pygame.

Como a responsabilidade de desenhar os objetos na Tela ficou com a classe KivyWorld e para que o usuário não precise mudar a classe que utiliza para representar o mundo ao selecionar o Kivy como *backend*, sobrescrevemos o método *show* da classe KivyCanvas, que era responsável por iniciar a janela do Pygame e então criamos um KivyWorld com os objetos do mundo criado pelo usuário e então o KivyWorld se encarrega de iniciar a janela do Kivy, através do *widget* e do App Kivy que são criados no construtor da classe KivyWorld.

Para que o Kivy seja usado pelo usuário como *backend*, basta ele chamar o método *set_backend* passando a *string 'kivy'* como parâmetro para que o *backend* utilizado seja o Kivy.

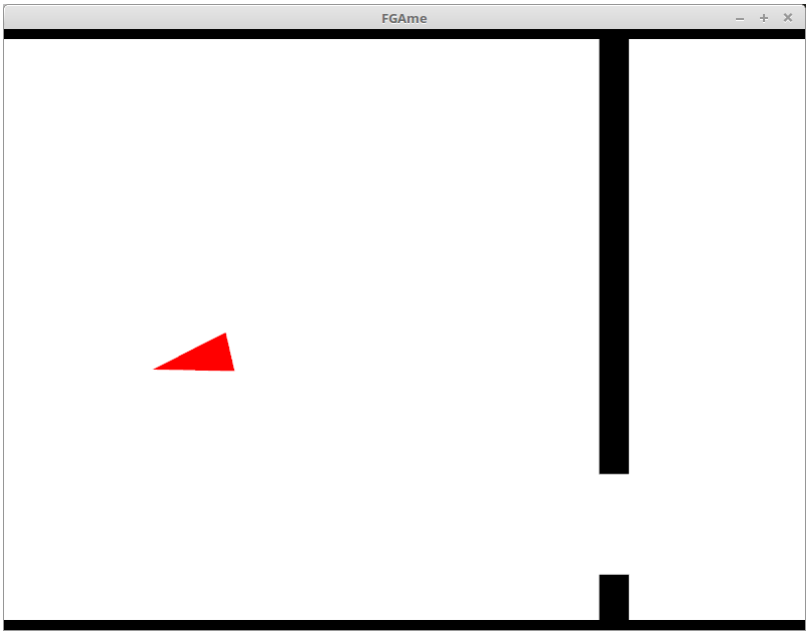

Figura 14 – Flappy Bird feita com a FGAme utilizando o Kivy como *backend*.

Para adicionar e remover objetos no KivyWorld depois do mundo ter sido criado, foi necessário utilizar a classe de sinais da FGAme. Modificamos a classe World para que quando um objeto fosse adicionado ou removido e o *backend* selecionado fosse o Kivy, é disparado um sinal que é tratado dentro da classe KivyWorld e então esse objeto é adicionado ou removido do App Kivy.

5.1.2 Refatoramento do *backend* Kivy

Foi decidido refatorar a solução feita do *backend* Kivy pela arquitetura de sinais que a FGAme possui implementada devido que a solução anterior basicamente tinha duas instâncias da World rodando simultaneamente, através das classes KivyWorld e World.

Criamos os seguintes sinais, *create_world_signal*, quando a instância de World é criada, é passado uma atributo em tempo de execução para a classe World com a instância da FGAmeWidget, onde todos os objetos serão adicionados e uma lista com os objetos kivy, para mapear as posições dos objetos da FGAme e do Kivy. Depois criamos os sinais *add_object_signal* e *remove_object_signal*, responsáveis por adicionar e remover os objetos do Widget. E por fim, criamos sinais responsáveis pela pausa da simulação e resumo da simulação e também da atualização das posições. Essa solução funciona muito bem pois os sinais são sempre disparados e são executados somente se o *backend* Kivy tiver sido selecionado pelo usuário, fazendo com que não seja necessário ter duas instâncias de World ao mesmo tempo.

5.2 Deploy para Android

Para conseguir fazer o *deploy* de um jogo feito com a FGAme no Android, utilizamos a biblioteca Buildozer. Como a FGAme depende do Python 3 para funcionar, é necessário instalar uma versão de desenvolvimento do Buildozer, pois a versão estável ainda não oferece suporte ao Python 3, somente o Python 2.

A versão utilizada da biblioteca foi a versão 0.34dev, e para o funcionamento da mesma, é necessário instalar a versão 0.23 do Cython, utilizando a versão mais recente do Cython, tivemos erros de compilação de alguns arquivos, os desenvolvedores do Buildozer recomendaram utilizar essa versão em algumas *issues* no repositório oficial.

Depois de instalar a versão de desenvolvimento do Buildozer de acordo com a documentação oficial, é necessário criar um arquivo chamado de *buildozer.spec* através do comando *buildozer init*, sendo que basta adicionar nesse arquivo o caminho do Crystax NDK, de acordo com a documentação oficial. Depois adicionar um arquivo no diretório chamado de *main.py*, para que seja criado o APK.

Primeiramente foi feito uma *build* do exemplo do jogo Pong, presente na documentação do Kivy, através do comando *buildozer android debug* e o APK foi gerado com sucesso.

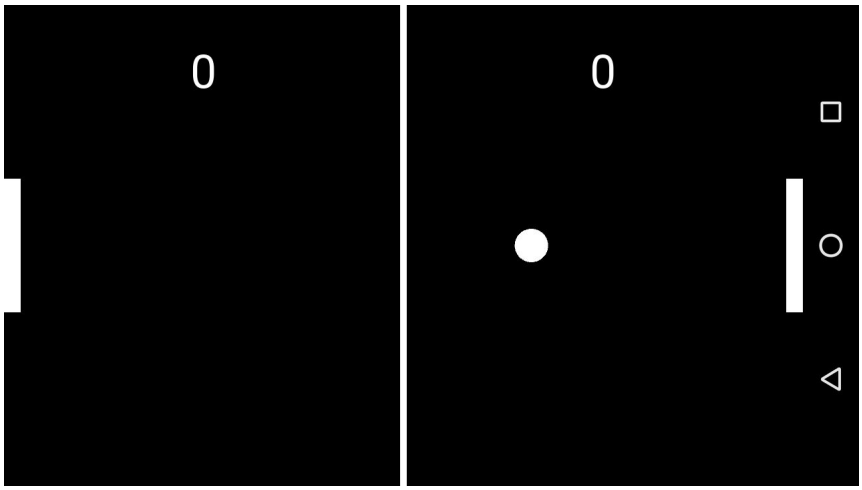

Figura 15 – Pong do Kivy utilizando Python 3 rodando no Android.

O próximo passo era conseguir integrar a FGAme com o Buildozer, para isso, editamos o arquivo *buildozer.spec* adicionando todas as dependências necessárias e então rodamos o comando *buildozer android debug* novamente e tivemos problemas de instalação das dependências. Depois de verificar algumas *issues* no repositório oficial do Buildozer, verificamos que o Buildozer tentava instalar as dependências utilizando a versão 2.7 do Python, para resolver esse problema, substituimos todas as ocorrências da *string* python2.7 com a *string* python3.5 dentro dos arquivos do diretório *.buildozer/* que é criado ao utilizar o comando *buildozer android debug*. Após fazer essa mudança, as dependências foram instaladas corretamente, sendo necessário substuir a pasta referente a FGAme dentro do diretório *android/platform/build/dists/myapp/crystax_python/crystax_python/site-packages/*, pois a versão da FGAme que se encontra no Pip não contém a implementação do *backend* Kivy.

Ao instalar o APK no Android, tivemos um problema de incompatibilidade com o Pillow, que é uma biblioteca de renderizar imagens que a FGAme utiliza, o erro apontado foi que um arquivo compilado dessa biblioteca era referente a um sistema de 64bits e o celular utilizado era de 32bits. Tivemos que adicionar um tratamento de exceção ao importar o Pillow, então atualmente não é possível utilizar o Pillow para os jogos rodarem no Android.

Depois de feito isso, foi gerado um novo APK, que funcinou corretamente, mas com um pequeno atraso devido ao carregamento do aplicativo no celular. A tela de *loading* do Kivy demorava muito para desaparecer e a simulação do jogo começava antes dessa tela de *loading* desaparecer para o usuário.

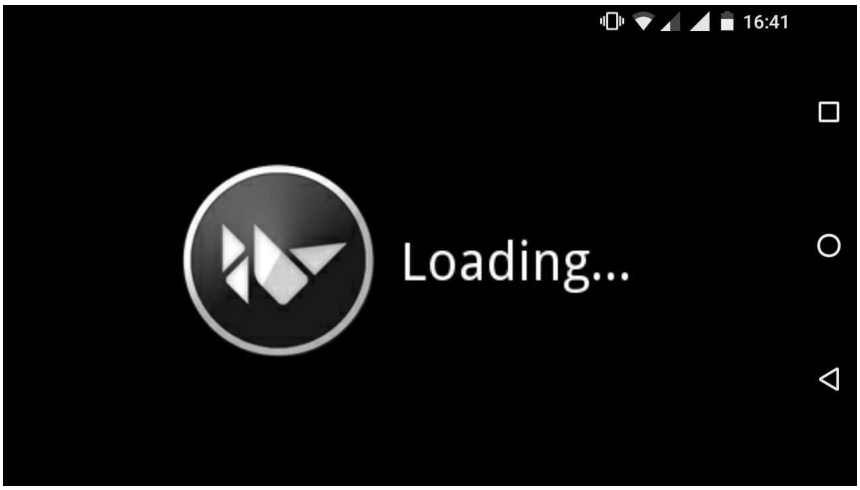

Figura 16 – Tela de *loading*.

Então foi necessário adicionar uma pequena pausa de 3 segundos na simulação para que o jogo não começasse antes da tela de *loading* desaparecer.

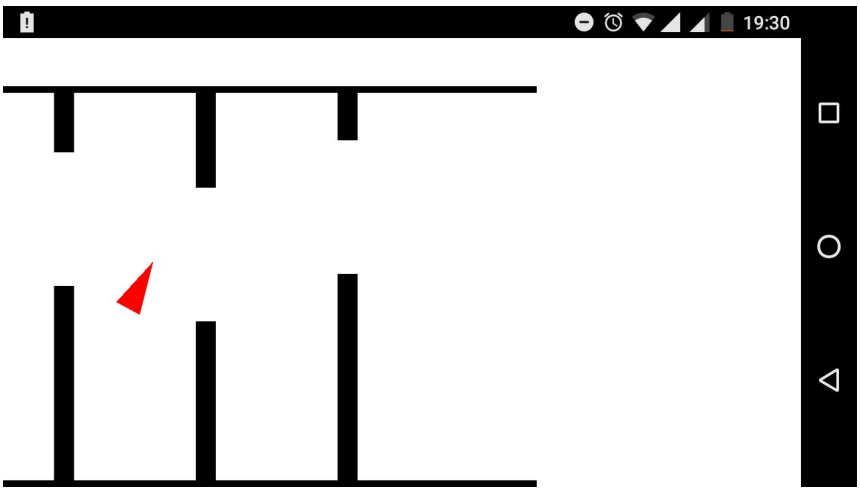

Figura 17 – Flappy Bird feito utilizando a FGAme rodando no Android.

5.3 Imagem Docker para geração de *build* do APK Android

Como o ambiente para gerar o APK Kivy é bastante sensível as diferentes versões das dependências, como Cython, o próprio Python, entre outras, foi desejável criar uma Imagem do Docker com o ambiente configurado do Buildozer, visto que a configuração desse ambiente foi bastante custosa.

A partir da imagem do Ubuntu 16.04, que foi o sistema operacional utilizado para geração da *build*, adicionamos todas as dependências do Buildozer e também o Crystax NDK no Dockerfile, que pode ser visualizado no Anexo A.

A partir do Dockerfile, criamos uma imagem Docker que se encontra disponível no Dockerhub, uma espécie de repositório do Docker onde várias imagens Docker são armazenadas. Para conseguir utilizar a imagem, basta ter o Docker instalado e rodar o comando abaixo.

```
$ docker pull gutioliveira/fgamebuildozer
```

Depois de baixar a Imagem Docker, é necessário criar um diretório novo contendo um arquivo chamado *main.py*, contendo o jogo criado com a FGAme e um arquivo *buildozer.spec*, contendo todas as dependências necessárias para instalar a FGAme dentro do *container* Docker. Deixamos no Anexo B o arquivo *buildozer.spec* necessário para geração do APK. Para gerar o APK, entre no diretório criado com os arquivos *main.py* e *buildozer.spec* e utilize o comando abaixo para acessar o *container* Docker.

```
$ docker run --rm -it --privileged -v $PWD:/src \
-v /dev/bus/usb:/dev/bus/usb -v \
/home/$USER:/home/kivy gutioliveira/fgamebuildozer
```

Como dito anteriormente, é necessário substituir as ocorrências das *strings* python2.7 por python3.5 dentro do arquivo *.buildozer*, então, dentro do *container* basta rodar o comando *builozer android debug* para criar o diretório *.buildozer*, depois rodar o comando abaixo para substituir as *strings* e então rodar o comando *builozer android debug* novamente para geração do APK Android.

```
$ find .buildozer/android/platform/python-for-
android-new-toolchain/pythonforandroid/
-type f -exec sed -i 's/python2.7/python3.5/g' {} \;
```

Após ter substituído as *strings*, não é mais necessário rodar o comando acima novamente toda vez que for gerar um novo APK.

6 Conclusão

O trabalho atingiu seu principal objetivo que era de apresentar a FGAme à comunidade acadêmica e também conseguiu cumprir seu principal objetivo de implementação que era conseguir rodar um jogo feito com a FGAme na plataforma Android sem ter que fazer alterações significativas no código fonte do jogo, bastando definir que o *backend* selecionado é o Kivy e utilizando o Buildozer para gerar o APK Android.

Esse foi o primeiro Trabalho de Conclusão de Curso em que a FGAme é tida como base, tivemos dificuldades de produzir um texto mais voltado para a acadêmia por conta de ser um tema bastante prático, então focamos mais em detalhar a FGAme, como foi feito o trabalho e por fim o que foi produzido. Com isso, acreditamos que este trabalho possa despertar o interesse de outras pessoas a contribuirem com a FGAme.

6.1 Trabalhos Futuros

Existem diversos trabalhos que podem ser realizados na FGAme e também melhorias no que foi produzido neste trabalho, como por exemplo:

- Ajuste automático da resolução de um jogo através da resolução do celular.
- Criação de um aplicativo Android que rode os jogos da FGAme, sem a necessidade que cada jogo seja um aplicativo distinto.
- Rodar um jogo da FGAme no iOS, utilizando o Kivy e Buildozer.

Outras funcionalidades que não envolvem o que foi feito neste trabalho também podem ser consideradas trabalhos futuros para a FGAme, como por exemplo:

- Melhorar a performance da FGAme com o Cython.
- Implementação de testes.
- Implementação de novos *backends*.
- Criação de jogos educacionais utilizando a FGAme.

Também identificamos algumas falhas na arquitetura da FGAme, que foram:

- Dependência excessiva do Pygame. A FGAme foi projetada tendo o Pygame como base, então as classes responsáveis pela coleta de *input* e de renderização funcionam da mesma maneira que o Pygame funciona.

- Classes muito acopladas. Uma possível solução seria utilizar a arquitetura de sinais da FGAme para diminuir o acoplamento.
- O Pygame não oferece nenhuma vantagem com relação ao Kivy e com o Kivy é possível rodar os jogos em um número maior de plataformas então o porte da FGAme completamente para o Kivy seja interessante.

Referências

2D Game Collision Detection: An introduction to clashing geometry in games. [S.l.]: Thomas Schwarzl, 2012. Citado na página 32.

BOGDANCHIKOV, A. et al. *Python to learn programming*. [S.l.]: IOPscience, 2013. Citado na página 19.

BOOPATHI, M. et al. Quantification of software code coverage using artificial bee colony optimization based on markov approach. 2017. Citado na página 47.

BOURG, D. M. *Physics for Game Developers*. [S.l.]: O'Reilly, 2002. Citado 3 vezes nas páginas 21, 27 e 29.

CYTHON. *Cython documentation*. [S.l.], 2018. Disponível em: <http://docs.cython.org/en/latest/src/quickstart/overview.html>. Citado na página 53.

ERICSON, C. *Real time collision detection*. [S.l.]: Morgan Kaufmann series in interactive 3D technology, 2005. Citado na página 27.

JASON, G. *Game Engine Architecture*. [S.l.]: Boca Raton: A K Peters, 2009. Citado na página 19.

KIVY. *Kivy - Open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps.* [S.l.], 2017. Disponível em: <https://kivy.org>. Citado na página 42.

LEI, H. et al. A statistical analysis of the effects of scrum and kanban on software development projects. 2008. Disponível em: <http://dl.acm.org/citation.cfm?doid=1593105.1593127>. Citado na página 51.

MEYER, M. Continuous integration and its tools. IEEE COMPUTER SOCIETY, 2014. Citado na página 49.

NUSSENZVEIG, H. M. *Curso de física básica 1 mecânica*. [S.l.]: Editora Edgard Blucher, 2002. Citado 4 vezes nas páginas 21, 22, 23 e 26.

OPENSOURCE.COM. *What is Docker?* [S.l.], 2018. Disponível em: <https://opensource.com/resources/what-docker>. Citado na página 54.

PYGAME. *Pygame is a Free and Open Source python programming language library for making multimedia applications like games built on top of the excellent SDL library.* [S.l.], 2017. Disponível em: <https://www.pygame.org>. Citado na página 42.

RABIN, S. *Software engineering A Practioner's Approach 7th edition*. [S.l.]: Higher Education, 2010. Citado 2 vezes nas páginas 47 e 48.

RADIGAN, D. How the kanban methodology applies to software development. Disponível em: <https://www.atlassian.com/agile/kanban>. Citado na página 51.

SDL. *Simple DirectMedia Layer is a cross-platform development library*. [S.l.], 2017. Disponível em: <https://www.libsdl.org/>. Citado na página 42.

SMITH, M. Towards modern literature programming. University of Canterburyn, 2001. Citado na página 48.

SOMMERVILLEN, I. *Software Documentation*. literateprogramming, 2001. Disponível em: <http://www.literateprogramming.com/documentation.pdf>. Citado 2 vezes nas páginas 47 e 48.

VAGAč, M.; KOLLáR, J. Improving program comprehension by automatic metamodel abstraction. Comput. Sci. Inf. Syst, 2012. Citado na página 48.

Anexos

ANEXO A – Dockerfile

```
FROM ubuntu:16.04

ENV DIR '/src'

ENV PATH /usr/local/bin:$PATH
ENV LANG C.UTF-8

ENV CYTHON_VERSION=0.25.2
ENV CRYSTAX_VERSION=10.3.2

RUN set -ex \
&& dpkg --add-architecture i386 \
&& apt-get update -y \
&& apt-get upgrade -y \
&& apt-get install -y build-essential libtool python-dev libportmidi-dev \
libswscale-dev libavformat-dev \
libavcodec-dev libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev \
libsdl2-ttf-dev python3-kivy python3-pip \
git unzip zlib1g-dev zlib1g:i386 \
openjdk-8-jdk libgtk2.0-0:i386 libpangox-1.0-0:i386 \
libpangoxft-1.0-0:i386 \
libidn11:i386 lib32stdc++6 libreadline-gplv2-dev \
libncursesw5-dev libssl-dev \
libsqlite3-dev tk-dev libgdbm-dev \
libc6-dev libbz2-dev wget \
libstdc++6:i386 bsdtar autotools-dev autoconf sudo

RUN set -ex \
&& update-alternatives --install /usr/bin/python python /usr/bin/python3.5 1 \
&& update-alternatives --install /usr/bin/pip pip /usr/bin/pip3 1 \
&& pip install Cython==$CYTHON_VERSION

RUN set -ex \
&& useradd kivy -mN \
&& echo "kivy:kivy" | chpasswd \
```

```
&& mkdir -p $DIR \
&& chown kivy:users /opt \
&& chown kivy:users /src

RUN set -ex \
&& sudo -u kivy -i \
&& cd /opt \
&& git clone https://github.com/kivy/buildozer \
&& cd buildozer \
&& python setup.py build \
&& pip install -e . \
&& sed -i -e 's/build.gradle/~build.gradle/g' /opt/buildozer/buildozer/targets/android.p

RUN set -ex \
&& sudo -u kivy -i \
&& cd /opt \
&& wget \
https://www.crystax.net/download/crystax-ndk-${CRYSTAX_VERSION}-linux-x86_64.tar.xz \
&& bsdtar xf crystax-ndk-${CRYSTAX_VERSION}-linux-x86_64.tar.xz \
&& rm crystax-ndk-${CRYSTAX_VERSION}-linux-x86_64.tar.xz

RUN ln -s /usr/local/bin/buildozer /bin/buildozer

RUN mkdir /buildozer && chown kivy /buildozer

WORKDIR $DIR

USER kivy
```

ANEXO B – buildozer.spec

```
[app]

# (str) Title of your application
title = FGAme App

# (str) Package name
package.name = myapp

# (str) Package domain (needed for android/ios packaging)
package.domain = org.test

# (str) Source code where the main.py live
source.dir = .

# (list) Source files to include (let empty to include all the files)
source.include_exts = py,png,jpg,kv,atlas

# (list) List of inclusions using pattern matching
#source.include_patterns = assets/*,images/*.png

# (list) Source files to exclude (let empty to not exclude anything)
#source.exclude_exts = spec

# (list) List of directory to exclude (let empty to not exclude anything)
#source.exclude_dirs = tests, bin

# (list) List of exclusions using pattern matching
#source.exclude_patterns = license,images/*/*.jpg

# (str) Application versioning (method 1)
version = 0.1

# (str) Application versioning (method 2)
# version.regex = __version__ = ['"](.*)['"]
# version.filename = %(source.dir)s/main.py
```

```
# (list) Application requirements
# comma seperated e.g. requirements = sqlite3,kivy
requirements = python3crystax,kivy,smallvectors==0.6.3,pygeneric==0.5.7,lazyutils==0.3.3

# (str) Custom source folders for requirements
# Sets custom source for any requirements with recipes

# (list) Garden requirements
#garden_requirements =

# (str) Presplash of the application
#presplash.filename = %(source.dir)s/data/presplash.png

# (str) Icon of the application
#icon.filename = %(source.dir)s/data/icon.png

# (str) Supported orientation (one of landscape, portrait or all)
orientation = landscape

# (list) List of service to declare
#services = NAME:ENTRYPOINT_TO_PY,NAME2:ENTRYPOINT2_TO_PY

#
# OSX Specific
#

#
# author = © Copyright Info

# change the major version of python used by the app
osx.python_version = 3

# Kivy version to use
osx.kivy_version = 1.9.1

#
# Android specific
#
```

```
# (bool) Indicate if the application should be fullscreen or not
fullscreen = 0

# (string) Presplash background color (for new android toolchain)
# Supported formats are: #RRGGBB #AARRGGBB or one of the following names:
# red, blue, green, black, white, gray, cyan, magenta, yellow, lightgray,
# darkgray, grey, lightgrey, darkgrey, aqua, fuchsia, lime, maroon, navy,
# olive, purple, silver, teal.
#android.presplash_color = #FFFFFF

# (list) Permissions
#android.permissions = INTERNET

# (int) Android API to use
#android.api = 19

# (int) Minimum API required
#android.minapi = 9

# (int) Android SDK version to use
#android.sdk = 20

# (str) Android NDK version to use
#android.ndk = 9c

# (bool) Use --private data storage (True) or --dir public storage (False)
#android.private_storage = True

# (str) Android NDK directory (if empty, it will be automatically downloaded.)
android.ndk_path = /opt/crystax-ndk-10.3.2

# (str) Android SDK directory (if empty, it will be automatically downloaded.)
#android.sdk_path =

# (str) ANT directory (if empty, it will be automatically downloaded.)
#android.ant_path =

# (bool) If True, then skip trying to update the Android sdk
```

```
# This can be useful to avoid excess Internet downloads or save time
# when an update is due and you just want to test/build your package
# android.skip_update = False

# (str) Android entry point, default is ok for Kivy-based app
#android.entrypoint = org.renpy.android.PythonActivity

# (list) Pattern to whitelist for the whole project
#android.whitelist =

# (str) Path to a custom whitelist file
#android.whitelist_src =

# (str) Path to a custom blacklist file
#android.blacklist_src =

# (list) List of Java .jar files to add to the libs so that pyjnius can access
# their classes. Don't add jars that you do not need, since extra jars can slow
# down the build process. Allows wildcards matching, for example:
# OUYA-ODK/libs/*.jar
#android.add_jars = foo.jar,bar.jar,path/to/more/*.jar

# (list) List of Java files to add to the android project (can be java or a
# directory containing the files)
#android.add_src =

# (list) Android AAR archives to add (currently works only with sdl2_gradle
# bootstrap)
#android.add_aars =

# (list) Gradle dependencies to add (currently works only with sdl2_gradle
# bootstrap)
#android.gradle_dependencies =

# (str) python-for-android branch to use, defaults to master
#p4a.branch = stable

# (str) OUYA Console category. Should be one of GAME or APP
# If you leave this blank, OUYA support will not be enabled
```

```
#android.ouya.category = GAME

# (str) Filename of OUYA Console icon. It must be a 732x412 png image.
#android.ouya.icon.filename = %(source.dir)s/data/ouya_icon.png

# (str) XML file to include as an intent filters in <activity> tag
#android.manifest.intent_filters =

# (list) Android additionnal libraries to copy into libs/armeabi
#android.add_libs_armeabi = libs/android/*.so
#android.add_libs_armeabi_v7a = libs/android-v7/*.so
#android.add_libs_x86 = libs/android-x86/*.so
#android.add_libs_mips = libs/android-mips/*.so

# (bool) Indicate whether the screen should stay on
# Don't forget to add the WAKE_LOCK permission if you set this to True
#android.wakelock = False

# (list) Android application meta-data to set (key=value format)
#android.meta_data =

# (list) Android library project to add (will be added in the
# project.properties automatically.)
#android.library_references =

# (str) Android logcat filters to use
#android.logcat_filters = *:S python:D

# (bool) Copy library instead of making a libpymodules.so
#android.copy_libs = 1

# (str) The Android arch to build for, choices: armeabi-v7a, arm64-v8a, x86
android.arch = armeabi-v7a

#
# Python for android (p4a) specific
#

# (str) python-for-android git clone directory (if empty, it will be automatically
```

```
#p4a.source_dir =

# (str) The directory in which python-for-android should look for your own build recipes
#p4a.local_recipes =

# (str) Filename to the hook for p4a
#p4a.hook =

# (str) Bootstrap to use for android builds
# p4a.bootstrap = sdl2


#
# iOS specific
#

# (str) Path to a custom kivy-ios folder
#ios.kivy_ios_dir = ../kivy-ios

# (str) Name of the certificate to use for signing the debug version
# Get a list of available identities: buildozer ios list_identities
#ios.codesign.debug = "iPhone Developer: <lastname> <firstname> (<hexstring>)"

# (str) Name of the certificate to use for signing the release version
#ios.codesign.release = %(ios.codesign.debug)s


[buildozer]

# (int) Log level (0 = error only, 1 = info, 2 = debug (with command output))
log_level = 2

# (int) Display warning if buildozer is run as root (0 = False, 1 = True)
warn_on_root = 1

# (str) Path to build artifact storage, absolute or relative to spec file
# build_dir = ./.buildozer

# (str) Path to build output (i.e. .apk, .ipa) storage
```

```
# bin_dir = ./bin

#    ----------------------------------------------------------------------------
#    List as sections
#
#    You can define all the "list" as [section:key].
#    Each line will be considered as a option to the list.
#    Let's take [app] / source.exclude_patterns.
#    Instead of doing:
#
#[app]
#source.exclude_patterns = license,data/audio/*.wav,data/images/original/*
#
#    This can be translated into:
#
#[app:source.exclude_patterns]
#license
#data/audio/*.wav
#data/images/original/*
#


#    ----------------------------------------------------------------------------
#    Profiles
#
#    You can extend section / key with a profile
#    For example, you want to deploy a demo version of your application without
#    HD content. You could first change the title to add "(demo)" in the name
#    and extend the excluded directories to remove the HD content.
#
#[app@demo]
#title = My Application (demo)
#
#[app:source.exclude_patterns@demo]
#images/hd/*
#
#    Then, invoke the command line with the "demo" profile:
#
#buildozer --profile demo android debug
```